HPX Documentation

pdf-docs

The STE||AR Group

March 04, 2019

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.

[1] https://github.com/STEllAR-GROUP/hpx/issues
[2] https://cpplang.slack.com
[3] hpx-users@stellar.cct.lsu.edu

CHAPTER 1

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 2

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 has 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[4] Group (**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch) at Louisiana State University (LSU)[5]'s Center for Computation and Technology (CCT)[6] to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STE||AR[7] is a research group focusing on system software solutions and scientific application development for hybrid and many-core hardware architectures.

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

For more information about the STE||AR[8] Group, see *People*.

2.1.3 What makes our systems slow?

Estimates say that we currently run our computers at way below 100% efficiency. The theoretical peak performance (usually measured in FLOPS[9]—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[10]) and weak scaling (see Gustafson's Law[11]). 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 can we 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 which limit scalability, mostly inherent to the hardware architectures and the programming models we use. We break these limitations into four fundamental factors which 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, 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[12], as published back in 1945, and later extended by the Harvard architecture[13]. These ideas form the foundation, the execution model of computer systems we use currently. But apparently 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:

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

- Performance: as mentioned, scalability and efficiency are the main criteria people are interested in
- Fault tolerance: the low expected mean time between failures (MTBF[14]) 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, even more so in the future
- Generality: any system should be usable for a broad set of use cases
- Programmability: for me as a 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, has 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 as well. However, we do not have the time to wait for a coherent system to evolve for another 50 years. 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 have already been had, often more than 30 years ago. 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, for instance low latency network technologies like InfiniBand[15], caching memory hierarchies in all modern processors, the constant optimization of existing MPI[16] implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use GPGPUs[17] 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.

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

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

if we already have special machines supporting this mode of operation, such as the Cray XMT[18]). 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)[19], Microsoft Parallel Patterns Library (PPL)[20], Cilk++[21], 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 barrier we mean the synchronization of the control flow between several (very often all) threads (when using OpenMP[22]) or processes (MPI[23]). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP[24] as the system synchronizes the threads used to execute the different iterations in parallel. In MPI[25] 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 acts as an eye of the 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 threads executing doing 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 could continue calculating other things, all you need is to have those iterations done which were producing the required results for a particular next operation. Good bye global barriers, hello constraint based synchronization! People have been trying to build this type of computing (and even computers) already back in the 1970's. 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[26] 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 overcome this limitation, such as Chapel[28], X10[29], UPC[30], or Fortress[31]. However all systems based on PGAS[32] rely

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

on static data distribution. This works fine as long as such a static data distribution does not result in homogeneous 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++[33]. The first attempts towards solving related problem go back decades as well, a good example is the Linda coordination language[34]. 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 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[35] 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) show 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 me look at the way we usually write our applications for clusters using MPI[36]. This programming model is all about data transfer between nodes. MPI[37] is the prevalent programming model for clusters, it is fairly straightforward to understand and to use. Therefore, we often write the applications in a way accommodating 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 are the overall machine utilization and the (strong) scaling characteristics. While it is not impossible to implement more dynamic, data driven, and asynchronous applications using MPI[38], it is overly difficult to so. At the same time, if we look at applications preferring to execute the code close the *locality* where the data was placed, i.e. utilizing active messages (for instance based on Charm++[39]), 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[40]. 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, any more than trivial MPI[41] application 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 are the adverse effects. The community has discovered message-driven and (data-driven) methods of implementing algorithms a long time ago, and systems such as Charm++[42] already have integrated active messages demonstrating the validity of the concept. Message driven computation allows sending messages without requiring the receiver to actively wait for them. Any incoming message is handled asynchronously and triggers the encoded action by passing along arguments and—possibly—continuations. *HPX* combines this scheme

[33] https://charm.cs.uiuc.edu/
[34] http://en.wikipedia.org/wiki/Linda_(coordination_language)
[35] http://en.wikipedia.org/wiki/GPGPU
[36] https://en.wikipedia.org/wiki/Message_Passing_Interface
[37] https://en.wikipedia.org/wiki/Message_Passing_Interface
[38] https://en.wikipedia.org/wiki/Message_Passing_Interface
[39] https://charm.cs.uiuc.edu/
[40] https://en.wikipedia.org/wiki/Message_Passing_Interface
[41] https://en.wikipedia.org/wiki/Message_Passing_Interface
[42] https://charm.cs.uiuc.edu/

with work queue-based scheduling as described above, which allows the system to overlap almost completely 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[43] 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 1.2.1
```

2.2.2 *HPX* dependencies

The minimum dependencies needed to use *HPX* are Boost[44] and Portable Hardware Locality (HWLOC)[45]. 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[46] and Portable Hardware Locality (HWLOC)[47], it is recommended that you don't use the system allocator, but instead use either `tcmalloc` from google-perftools[48] (default) or jemalloc[49] 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
```

43 https://stellar-group.org/downloads/
44 https://www.boost.org/
45 https://www.open-mpi.org/projects/hwloc/
46 https://www.boost.org/
47 https://www.open-mpi.org/projects/hwloc/
48 https://code.google.com/p/gperftools
49 https://www.canonware.com/jemalloc

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[50] 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[51] where to find them by passing one or more of the following options to CMake[52]:

```
-DBOOST_DIR=/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[53].

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 `simplest_hello_world_1` example by running:

```
make simplest_hello_world_1
```

HPX executables end up in the `bin` directory in your build directory. You can now run `simplest_hello_world_1` and should see the following output:

```
./bin/simplest_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/simplest_hello_world_1.cpp`. The `hello_world` 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

[50] https://www.cmake.org
[51] https://www.cmake.org
[52] https://www.cmake.org
[53] https://www.cmake.org

available targets. For example, `make help | grep hello_world` outputs the following:

```
... examples.quickstart.simplest_hello_world_2
... simplest_hello_world_2
... examples.quickstart.simplest_hello_world_1
... simplest_hello_world_1
... hello_world
... examples.quickstart.hello_world
```

It is also possible to build e.g. all quickstart examples using `make examples.quickstart`.

2.2.4 Hello, World!

The following `CMakeLists.txt` is a minimal example of what you need in order to build an executable using CMake[54] and *HPX*:

```
cmake_minimum_required(VERSION 3.3.2)
project(my_hpx_project CXX)
find_package(HPX REQUIRED)
add_hpx_executable(my_hpx_program
    SOURCES main.cpp
    COMPONENT_DEPENDENCIES iostreams)
```

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: `COMPONENT_DEPENDENCIES iostreams` 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/include/iostreams.hpp>

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

Then, in your project directory run the following:

```
mkdir build && cd build
cmake -DCMAKE_PREFIX_PATH=/path/to/hpx/installation ..
make all
./my_hpx_program
```

54 https://www.cmake.org

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
//
//  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.

//[simplest_hello_world_2_getting_started
#include <hpx/hpx_init.hpp>
#include <hpx/include/iostreams.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.

2.2.5 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/iostreams.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/include/parallel_generate.hpp>
#include <hpx/include/parallel_sort.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;
}

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::parallel::generate(hpx::parallel::execution::seq,
                  std::begin(v), std::end(v), &std::rand);

    // We can launch the sort in parallel and asynchronously.
    hpx::future<void> done_sorting =
        hpx::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;

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

2.2.6 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[55] models. Partitioned Global Address Space is passive in their means of address translation. Copy semantics, distributed compound operations, and affinity relationships are some of the global functionality supported by AGAS.

Process The concept of the "process" in *HPX* is extended beyond that of either sequential execution or communicating sequential processes. While the notion of process suggests action (as do "function" or "subroutine") it has a further responsibility of context, that is, the logical container of program state. It is this aspect of operation that process is employed in *HPX*. Furthermore, referring to "parallel processes" in *HPX* designates the presence of 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[56], the J-machine's[57] 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.

[55] https://www.pgas.org/
[56] http://en.wikipedia.org/wiki/Dataflow_architecture
[57] http://en.wikipedia.org/wiki/J%E2%80%93Machine

A parcel can cause actions to occur remotely and asynchronously, among which are the creation of threads at different system nodes or synchronous domains.

Local Control Object

Lightweight Control Object

LCO A local control object (sometimes called a lightweight control object) is a general term for the synchronization mechanisms used in *HPX*. Any object implementing a certain concept can be seen as an LCO. This concepts encapsulates the ability to be triggered by one or more events which when taking the object into a predefined state will cause a thread to be executed. This could either create a new thread or resume an existing thread.

The LCO is a family of synchronization functions potentially representing many classes of synchronization constructs, each with many possible variations and multiple instances. The LCO is sufficiently general that it can subsume the functionality of conventional synchronization primitives such as spinlocks, mutexes, semaphores, and global barriers. However due to the rich concept an LCO can represent powerful synchronization and control functionality not widely employed, such as dataflow and futures (among others), which open up enormous opportunities for rich diversity of distributed control and operation.

See *Using LCOs* for more details on how to use LCOs in *HPX*.

Action An action is a function that can be invoked remotely. In *HPX* a plain function can be made into an action using a macro. See *Applying actions* for details on how to use actions in *HPX*.

Component A component is a C++ object which can be accessed remotely. A component can also contain member functions which can be invoked remotely. These are referred to as component actions. See *Writing components* for details on how to use components in *HPX*.

2.4 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. 2.1 below, a future encapsulates a delayed computation. It acts as a proxy for a result initially not known, most of the time because the computation of the result has not completed yet. The future synchronizes the access of this value by optionally suspending any *HPX*-threads requesting the result until the value is available. When a future is created, it spawns a new *HPX*-thread (either remotely with a *parcel* or locally by placing it into the thread queue) which, when run, will execute the function associated with the future. The arguments of the function are bound when the future is created.

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.

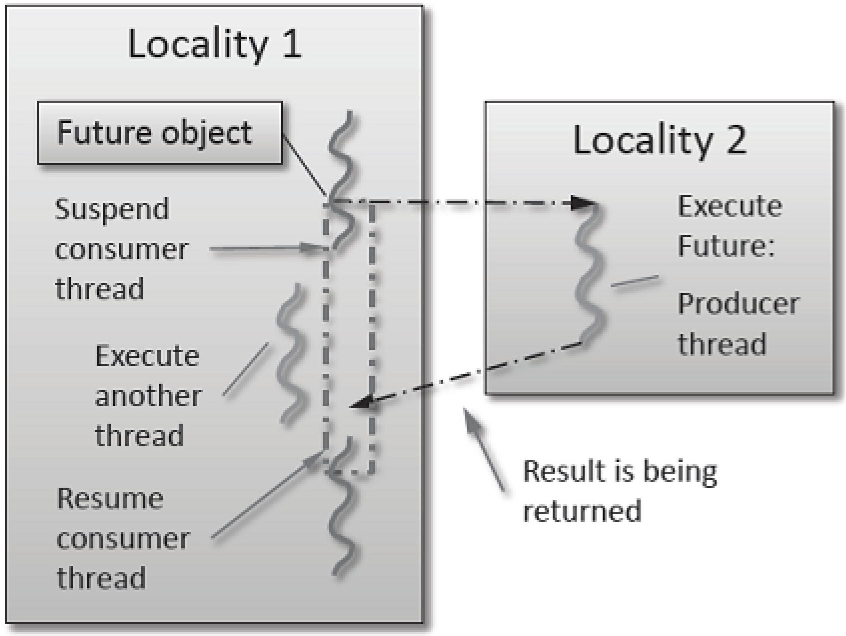

Fig. 2.1: Schematic of a future execution.

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[58] 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 `boost::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[59] 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
    boost::program_options::options_description
       desc_commandline("Usage: " HPX_APPLICATION_STRING " [options]");

    desc_commandline.add_options()
        ( "n-value",
          boost::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(boost::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());
    }
```

[58] https://www.boost.org/doc/html/program_options.html
[59] https://www.boost.org/doc/

```
    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*:

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

    desc_commandline.add_options()
        ( "n-value",
          boost::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.

```
int hpx_main(boost::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* that we wish to calculate. To further understand this we turn to the code to find where `fibonacci_action` was defined:

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

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

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

```
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.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
```

To run the program type:

```
./bin/hello_world
```

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 --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,
    // 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();

    // Find the global name of the current locality.
    hpx::naming::id_type const here = hpx::find_here();

```

```
    // 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.
            typedef hello_world_worker_action action_type;
            futures.push_back(hpx::async<action_type>(here, 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. `hpx::lcos::wait_each` for each one. 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())
            << hpx::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);
}

// Define the boilerplate code necessary for the function 'hello_world_worker'
// to be invoked as an HPX action (by a HPX future). This macro defines the
// type 'hello_world_worker_action'.
HPX_PLAIN_ACTION(hello_world_worker, hello_world_worker_action);
```

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*; accumulator.

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
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 components functionality.
- **A client class** - A high-level interface that acts as a proxy for an instance of the component.

Typically, these two classes all 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, 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.
            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

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

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

HPX_REGISTER_COMPONENT(accumulator_type, accumulator);

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

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

The client class

The following code is from accumulator.hpp.

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

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

and to invoke component actions:

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

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

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

For readability, we typedef the base class like so:

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

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

There are a few different ways of invoking actions:

- **Non-blocking**: For actions which 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;
    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()` (e.g., 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 which allow users to understand and use them easily. Dataflows, being a *LCO*, is triggered when the values it depends on become available. For instance, if you have a calculation X that depends on the result 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 which 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.

Nevertheless, we have chosen for the sake of example 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.

```
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]")
    ;

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

Next we look at 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();

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.6 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. 2.2 below.

We parallelize this code over the following eight examples:

- `Example 1`
- `Example 2`
- `Example 3`
- `Example 4`

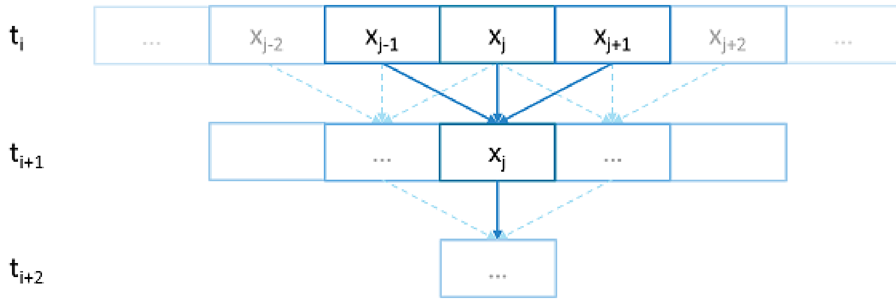

Fig. 2.2: Heat diffusion example program flow.

- `Example 5`
- `Example 6`
- `Example 7`
- `Example 8`

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

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

    // Our data for one time step
    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, are 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 which is serially executed into a code which creates asynchronous threads. In the simplest case this involves replacing a variable with a future to a variable, a function with a future to a function, and adding a `.get()` at the point where a value is actually needed. The code below shows how this technique was applied to the `struct stepper`.

```
struct stepper
{
    // Our partition type
    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 when as soon as its dependencies
        // are ready and hardware is available.

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

In example 2, we re-define 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 which 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 `unwrapped`, 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 which 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 contained in each partition (and therefore how many data points that are represented by one future) a user can now control the grainsize of the simulation. The rest of the code was then futurized in the same manner that was done in 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 which 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 neighboring partition, which could be remote, we must retrieve the data from 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 is passed 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 then to the `unwrapped` 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 in 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 the partition constructor.

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

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

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

```
    // The partitioned operator, it invokes the heat operator above on all elements
    // of a partition.
    static partition heat_part(partition const& left,
        partition const& middle, partition const& right)
    {
        using hpx::dataflow;
        using hpx::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)
                        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(), 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 which 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 a 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[60] 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[61].

2.5.2 *HPX* build system

The build system for *HPX* is based on CMake[62]. 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[63] 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[64] scripts try to adhere to the general cmake policies on how to write CMake[65] based projects.

Basic CMake usage

This section explains basic aspects of CMake, mostly for explaining those options which you may need on your day-to-day usage.

[60] https://stellar-group.org/downloads/
[61] https://stellar-group.org/downloads/
[62] https://www.cmake.org
[63] https://www.cmake.org
[64] https://www.cmake.org
[65] https://www.cmake.org

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 requires to know for which build tool it shall generate files (GNU make, Visual Studio, Xcode, etc.). If not specified on the command line, it tries to guess it based on you environment. Once identified the build tool, CMake uses the corresponding Generator for creating files for your build tool. You can explicitly specify the generator with the command line option `-G "Name of the generator"`. For knowing 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 9 2008" path/to/hpx
```

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

We use here the command-line, non-interactive CMake[66] interface.

1. Download and install CMake here: CMake Downloads[67]. Version 3.3.2 is the minimally 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. It is not supported to build *HPX* on the source directory. 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[68] for a detailed overview of specific CMake[69] variables.

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

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. It is correct to write the variable and the type on the CMake command line:

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

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	Recommended Version	Minimum Version	Architectures
Linux	3.2	2.6	x86-32, x86-64, k1om
BlueGeneQ	V1R2M0	V1R2M0	PowerPC A2
Windows	7, Server 2008 R2	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[70] and Portable Hardware Locality (HWLOC)[71]. So, before you read further, please make sure you have a recent version of Boost[72] installed on your target machine. *HPX* currently requires at least Boost V1.58.0 to work properly. It may build and run with older versions, but we do not test *HPX* with those versions, so please be warned.

Installing the Boost libraries is described in detail in Boost's own Getting Started document. 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.

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

70 https://www.boost.org/
71 https://www.open-mpi.org/projects/hwloc/
72 https://www.boost.org/

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[73].

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

Name	Recommended version	Minimum version	Notes
Compilers			
GNU Compiler Collection (g++)[74]	4.9 or newer	4.9	
Intel Composer XE Suites[75]	2014 or newer	2014	
clang: a C language family frontend for LLVM[76]	3.8 or newer	3.8	
Build System			
CMake[77]	3.9.0	3.3.2	Cuda support 3.9
Required Libraries			
Boost C++ Libraries[78]	1.62.0 or newer	1.58.0	
Portable Hardware Locality (HWLOC)[79]	1.11	1.2 (Xeon Phi: 1.6)	

Note: When compiling *HPX* using clang/libc++ on OSX platform it is advised not to use Boost V1.58 or V1.60.

Note: When compiling with the Intel Compiler on Linux systems, we only support C++ Standard Libraries provided by gcc 4.8 and upwards. If the `g++` in your path is older than 4.8, please specify the path of a newer `g++` by setting `CMAKE_CXX_FLAGS='-gxx-name=/path/to/g++'` via CMake[80].

Note: When building Boost using gcc please note that it is always a good idea to specify a `cxxflags=-std=c++11` command line argument to `b2` (`bjam`). Note however, that this is absolutely necessary when using gcc V5.2 and above.

73 https://rostam.cct.lsu.edu/
74 https://gcc.gnu.org
75 https://software.intel.com/en-us/intel-composer-xe/
76 https://clang.llvm.org/
77 https://www.cmake.org
78 https://www.boost.org/
79 https://www.open-mpi.org/projects/hwloc/
80 https://www.cmake.org

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

Name	Recommended version	Minimum version	Notes
Compilers			
Visual C++[81] (x64)	2015	2015	
Build System			
CMake[82]	3.9.0	3.3.2	
Required Libraries			
Boost[83]	1.62.0 or newer	1.58.0	
Portable Hardware Locality (HWLOC)[84]	1.11	1.5	

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

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

Name	Recommended version	Minimum version	Notes
google-perftools[85]	1.7.1	1.7.1	Used as a replacement for the system allocator, and for allocation diagnostics.
libunwind[86]	0.99	0.97	Dependency of google-perftools on x86-64, used for stack unwinding.
Open MPI[87]	1.10.1	1.8.0	Can be used as a highspeed communication library backend for the parcelport.

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

Name	Recommended version	Minimum version	Notes
Performance Application Programming Interface (PAPI)	Used for accessing hardware performance data.		
jemalloc[88]	2.1.2	2.1.0	Used as a replacement for the system allocator.
Hierarchical Data Format V5 (HDF5)[89]	1.8.7	1.6.7	Used for data I/O in some example applications. See important note below.

[81] https://msdn.microsoft.com/en-us/visualc/default.aspx
[82] https://www.cmake.org
[83] https://www.boost.org/
[84] https://www.open-mpi.org/projects/hwloc/
[85] https://code.google.com/p/gperftools
[86] https://www.nongnu.org/libunwind
[87] https://www.open-mpi.org

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

Name	Recommended version	Minimum version	Notes
Hierarchical Data Format V5 (HDF5)[90]	1.8.7	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=""'
```

Documentation

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

- `python` (2 or 3)
- `sphinx` (Python package)
- `sphinx_rtd_theme` (Python package)
- `breathe` (Python package)
- `doxygen`

If the Python[91] dependencies are not available through your system package manager you can install them using the Python[92] package manager `pip`:

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

You may need to set the following CMake[93] variables to make sure CMake[94] can find the required dependencies.

DOXYGEN_ROOT:PATH
Specifies where to look for the installation of the Doxygen[95] tool.

SPHINX_ROOT:PATH
Specifies where to look for the installation of the Sphinx[96] tool.

BREATHE_APIDOC_ROOT:PATH
Specifies where to look for the installation of the Breathe[97] tool.

88 https://www.canonware.com/jemalloc
89 https://www.hdfgroup.org/HDF5
90 https://www.hdfgroup.org/HDF5
91 https://www.python.org
92 https://www.python.org
93 https://www.cmake.org
94 https://www.cmake.org
95 https://www.doxygen.org
96 http://www.sphinx-doc.org
97 https://breathe.readthedocs.io/en/latest

Installing Boost

Important: When building Boost using gcc please note that it is always a good idea to specify a `cxxflags=-std=c++11` command line argument to `b2` (`bjam`). Note however, that this is absolutely necessary when using gcc V5.2 and above.

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 be also 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[98]. 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 what platform you are on the basic instructions are as follows (with possible additional platform-dependent command line arguments):

```
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[99]. 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
```

[98] https://www.boost.org/users/download/
[99] https://www.open-mpi.org/software/hwloc/v1.11

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[100] 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).
  - `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 exposed memory based performance counters (only available on Linux).
  - `io_counter_component`: A dynamically loaded plugin 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, this 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 *Platform specific build recipes* which show some basic steps on how to build HPX for a specific platform.

[100] https://www.open-mpi.org/projects/hwloc/doc/

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[101] 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 minimal 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[102] variables:

- `HPX_WITH_CXX0X`: Enables Pre-C++11 support (This is the minimal required mode on older gcc versions).
- `HPX_WITH_CXX11`: Enables C++11 support
- `HPX_WITH_CXX14`: Enables C++14 support
- `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[103]). Available build types are:

- **Debug**: Full debug symbols available and 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[104] builds, this means that the `CMAKE_BUILD_TYPE` variables have to match and for projects not using CMake[105], the `HPX_DEBUG` macro has to be set in debug mode.

Platform specific notes

Some platforms require to have special link and/or compiler flags specified to build HPX. This is handled via CMake[106]'s support for different toolchains (see cmake-toolchains(7)[107] 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.

[101] https://www.cmake.org
[102] https://www.cmake.org
[103] https://cmake.org/cmake/help/latest/variable/CMAKE_BUILD_TYPE.html
[104] https://www.cmake.org
[105] https://www.cmake.org
[106] https://www.cmake.org
[107] https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html

In order to enable full static linking with the libraries, the CMake[108] 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 the these handlers and the stack trace will not point to the location where the unhandled exception was thrown.

In general, core files are a helpful tool to inspect the state of the application at the moment of the crash (post-mortem debugging), without the need of attaching a debugger beforehand. This approach to debugging is especially useful if the error cannot be reliably reproduced, as only a single crashed application run is required to gain potentially helpful information like a stacktrace.

To debug with core files, the operating system first has to be told to actually write them. On most unix systems this can be done by calling:

```
ulimit -c unlimited
```

in the shell. Now the debugger can be started up with:

```
gdb <application> <core file name>
```

The debugger should now display the last state of the application. The default file name for core files is `core`.

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:

[108] https://www.cmake.org

```
cmake -DBOOST_ROOT=~/packages/boost -DHWLOC_ROOT=/packages/hwloc -DCMAKE_INSTALL_
↪PREFIX=~/packages/hpx ~/downloads/hpx_0.9.10
```

- 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 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.)

You can then use as build command either:

```
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 include prefix you had 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 3.2, execute:

```
cmake ../hpx \
    -DCMAKE_CXX_COMPILER=clang++ \
    -DBOOST_INCLUDE_DIR=/usr/local/include/boost-1_53 \
    -DBOOST_LIBRARY_DIR=/usr/local/lib \
    -DBOOST_SUFFIX=-clang-darwin32-mt-1_53 \
make
```

To build with Clang 3.3 (trunk), execute:

```
cmake ../hpx \
    -DCMAKE_CXX_COMPILER=clang++ \
    -DBOOST_INCLUDE_DIR=/usr/local/include/boost-1_53 \
    -DBOOST_LIBRARY_DIR=/usr/local/lib \
    -DBOOST_SUFFIX=-clang-darwin33-mt-1_53 \
make
```

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

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 4.8, and hwloc:

   ```
   sudo port install gcc48
   sudo port install hwloc
   ```

   You may also want:

```
sudo port install cmake
sudo port install git-core
```

3. Make this version of gcc your default compiler:

```
sudo port install gcc_select
sudo port select gcc mp-gcc48
```

4. 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 http://sourceforge.net/projects/boost/files/boost/1.54.0/boost_1_54_0.tar.bz2
tar xjf boost_1_54_0.tar.bz2
pushd boost_1_54_0
export BOOST_ROOT=$HOME/boost_1_54_0
./bootstrap.sh --prefix=$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=$HOME/hpx
cmake -DCMAKE_C_COMPILER=gcc \
    -DCMAKE_CXX_COMPILER=g++ \
    -DCMAKE_FORTRAN_COMPILER=gfortran \
    -DCMAKE_C_FLAGS="-Wno-unused-local-typedefs" \
    -DCMAKE_CXX_FLAGS="-Wno-unused-local-typedefs" \
    -DBOOST_ROOT=$BOOST_ROOT \
    -DHWLOC_ROOT=/opt/local \
    -DCMAKE_INSTALL_PREFIX=$HOME/hpx \
         $(pwd)/../hpx
make -j8
make -j8 install
export DYLD_LIBRARY_PATH=$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. 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 e.g. with 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[109]
- 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[110].
- Download the latest version of *HPX* from the STE||AR website: *HPX* Downloads[111].

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 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[112] and Portable Hardware Locality (HWLOC)[113] 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 red anymore).
- Press "Generate".
- Open up the build folder, and double-click hpx.sln.

[109] https://www.boost.org/users/download/
[110] https://www.cmake.org/cmake/resources/software.html
[111] https://stellar-group.org/downloads/
[112] https://www.boost.org/
[113] https://www.open-mpi.org/projects/hwloc/

- Build the INSTALL target.

For more detailed information about using CMake[114] 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[115] V3.4.3 installer (or latest version) from here[116]
- Download the Portable Hardware Locality (HWLOC)[117] V1.11.0 (or latest version) from here[118] and unpack it.
- Download Boost[119] libraries V1.60 (or latest version) from here[120] and unpack them.
- Build the boost DLLs and LIBs by using these commands from Command Line (or PowerShell). Open CMD/PowerShell inside the Boost dir and type in:

  ```
  bootstrap.bat
  ```

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

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

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

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

  Inside the '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 process will happen. This is important because the building machinery will do an "out-of-tree" build. CMake is not touching or changing in any way the original source files. Instead, it will generate Visual Studio Solution Files which will build HPX packages out of the HPX source tree.

- Set four new environment variables (in CMake, not in Windows environment, by the way): `BOOST_ROOT`, `HWLOC_ROOT`, `CMAKE_INSTALL_PREFIX` and `HPX_WITH_BOOST_ALL_DYNAMIC_LINK`. The meaning of these variables is as follows:
  - `BOOST_ROOT` the root directory of the unpacked Boost headers/cpp files.
  - `HWLOC_ROOT` the root directory of the unpacked Portable Hardware Locality files.
  - `CMAKE_INSTALL_PREFIX` the "root directory" where the future builds of HPX should be installed to.

  Note: HPX is a BIG software collection and I really don't recommend using the default `C:\Program Files\hpx`. I prefer simpler paths *without* white space, 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 "Path" text field. Repeat this for the first three variables.

[114] https://www.cmake.org
[115] https://www.cmake.org
[116] https://blog.kitware.com/cmake-3-4-3-available-for-download/
[117] https://www.open-mpi.org/projects/hwloc/
[118] http://www.open-mpi.org/software/hwloc/v1.11/downloads/hwloc-win64-build-1.11.0.zip
[119] https://www.boost.org/
[120] https://sourceforge.net/projects/boost/files/boost/1.60.0/

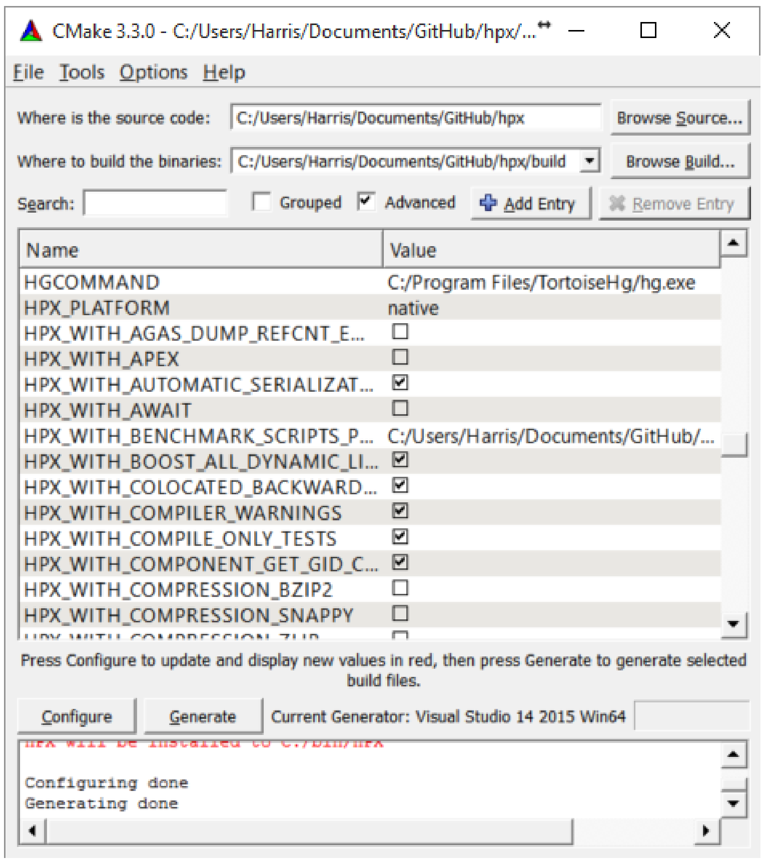

Fig. 2.3: Example CMake path settings.

The last one: `HPX_WITH_BOOST_ALL_DYNAMIC_LINK` is a `BOOL` and must be checked (there will be a checkbox instead of a textfield).

This is how variable insertion looks like:

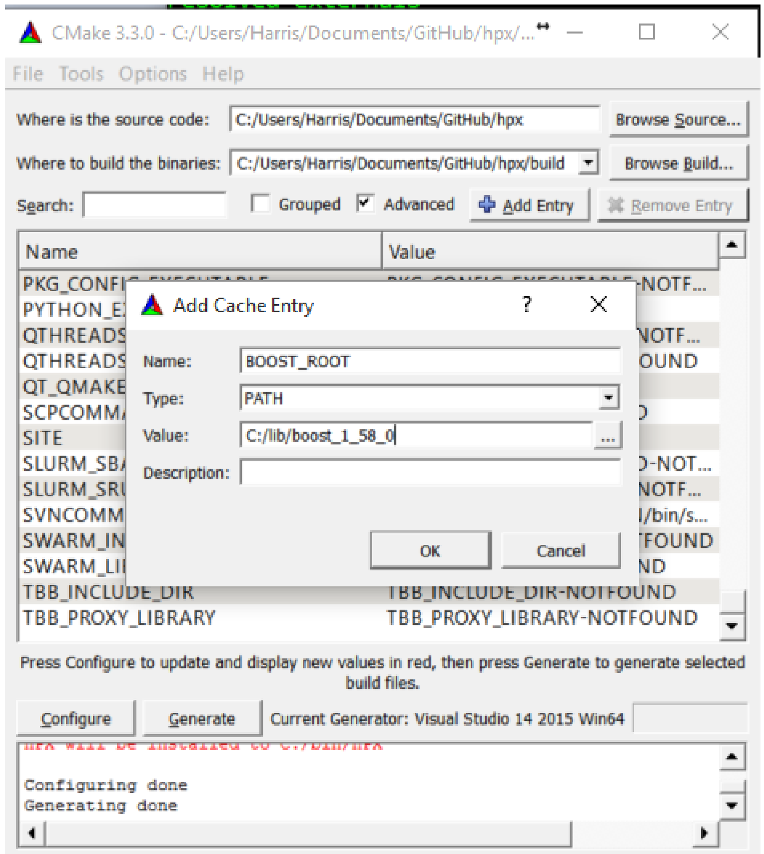

Fig. 2.4: Example CMake adding entry.

Alternatively you could provide `BOOST_LIBRARYDIR` instead of `BOOST_ROOT` with a difference that `BOOST_LIBRARYDIR` should point to the subdirectory inside Boost root where all the compiled DLLs/LIBs are. I myself have used `BOOST_LIBRARYDIR` which pointed to the `bin.v2` subdirectory under the Boost rootdir. Important is 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 Boost root folder where the compiled binaries are.

- Click the 'Configure' button of CMake-GUI. You will be immediately presented a small window where you can select the C++ compiler to be used within Visual Studio. In my case I have used 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:

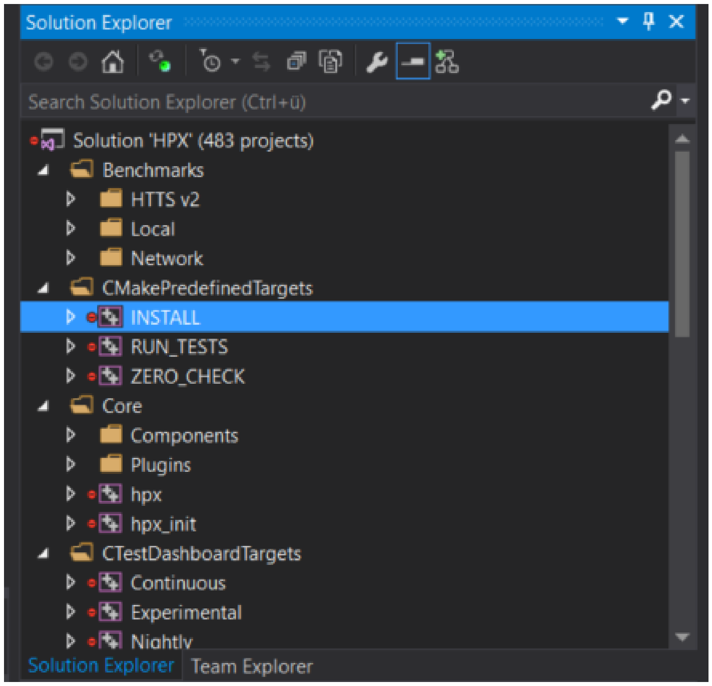

Fig. 2.5: Visual Studio INSTALL target.

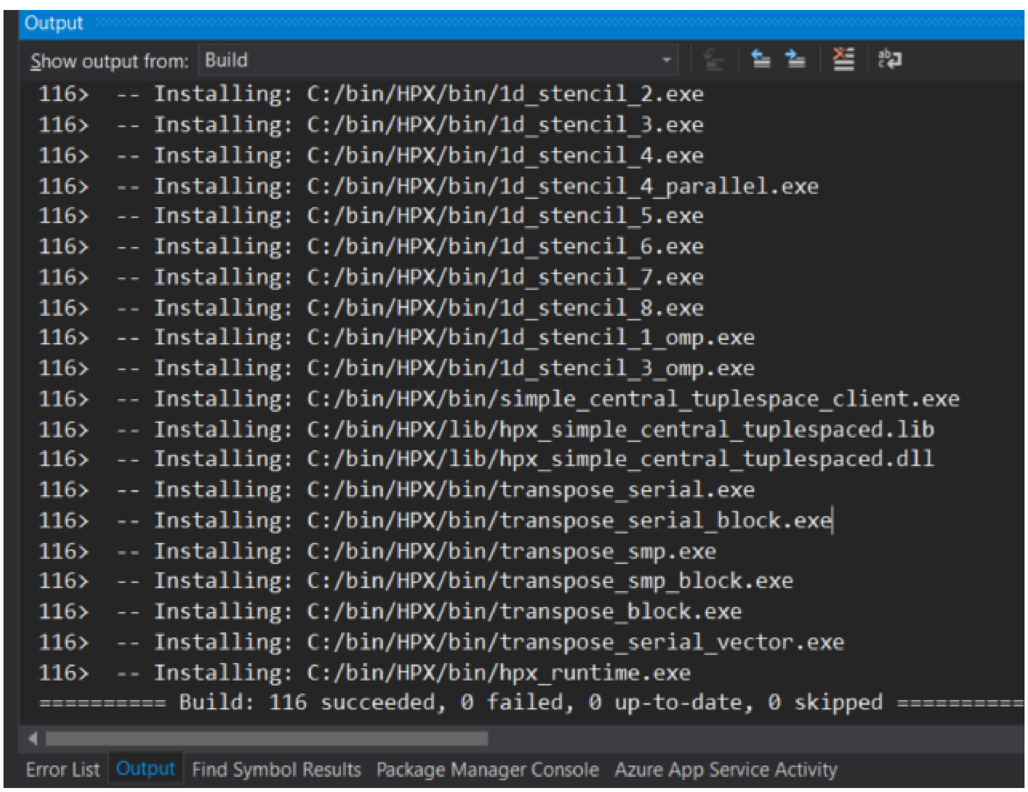

Fig. 2.6: Visual Studio build output.

How to Install *HPX* on BlueGene/Q

So far we only support BGClang for compiling *HPX* on the BlueGene/Q.

- Check if BGClang is available on your installation. If not obtain and install a copy from the BGClang trac page[121].
- Build (and install) a recent version of Hwloc Downloads[122]. With the following commands:

```
./configure \
  --host=powerpc64-bgq-linux \
  --prefix=$HOME/install/hwloc \
  --disable-shared \
  --enable-static \
  CPPFLAGS='-I/bgsys/drivers/ppcfloor -I/bgsys/drivers/ppcfloor/spi/include/
↪kernel/cnk/'
make
make install
```

- Build (and install) a recent version of Boost, using BGClang. To build Boost with BGClang, you'll need to set up the following in your Boost `~/user-config.jam` file:

```
# user-config.jam (put this file into your home directory)
using clang
  :
  : bgclang++11
  :
  ;
```

  You can then use this as your build command:

```
./bootstrap.sh
./b2 --build-dir=/tmp/build-boost --layout=versioned toolset=clang -j12
```

- Clone the master *HPX* git repository (or a stable tag):

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

- Generate the *HPX* buildfiles using cmake:

```
cmake -DHPX_PLATFORM=BlueGeneQ \
        -DCMAKE_TOOLCHAIN_FILE=/path/to/hpx/cmake/toolchains/BGQ.cmake \
        -DCMAKE_CXX_COMPILER=bgclang++11 \
        -DMPI_CXX_COMPILER=mpiclang++11 \
        -DHWLOC_ROOT=/path/to/hwloc/installation \
        -DBOOST_ROOT=/path/to/boost \
        -DHPX_WITH_MALLOC=system \
        /path/to/hpx
```

- To complete the build and install *HPX*:

```
make -j24
make install
```

  This will build and install the essential core components of *HPX* only. Use:

[121] https://trac.alcf.anl.gov/projects/llvm-bgq
[122] https://www.open-mpi.org/software/hwloc/v1.11

```
make -j24 examples
make -j24 install
```

to build and install the examples.

How to Install *HPX* on the Xeon Phi

Installation of the Boost Libraries

- Download Boost Downloads[123] for Linux and unpack the retrieved tarball.
- Adapt your `~/user-config.jam` to contain the following lines:

```
## Toolset to be used for compiling for the host
using intel
    : host
    :
    : <cxxflags>"-std=c++0x"
    ;

## Toolset to be used for compiling for the Xeon Phi
using intel
    : mic
    :
    : <cxxflags>"-std=c++0x -mmic"
      <linkflags>"-std=c++0x -mmic"
    ;
```

- Change to the directory you unpacked boost in (from now on referred to as `$BOOST_ROOT`) and execute the following commands:

```
./bootstrap.sh
./b2 toolset=intel-mic -j<N>
```

You should now have all the required boost libraries.

Installation of the Hwloc library

- Download Hwloc Downloads[124], unpack the retrieved tarball and change to the newly created directory.
- Run the configure-make-install procedure as follows:

```
CC=icc CFLAGS=-mmic CXX=icpc CXXFLAGS=-mmic LDFLAGS=-mmic ./configure --host=x86_
↪64-k1om-linux --prefix=$HWLOC_ROOT
make
make install
```

Important: The minimally required version of the Portable Hardware Locality (HWLOC) library on the Intel Xeon Phi is V1.6.

You now have a working hwloc installation in `$HWLOC_ROOT`.

[123] https://www.boost.org/users/download/
[124] https://www.open-mpi.org/software/hwloc/v1.11

Building *HPX*

After all the prerequisites have been successfully installed, we can now start building and installing *HPX*. The build procedure is almost the same as for *How to install HPX on Unix variants* with the sole difference that you have to enable the Xeon Phi in the CMake Build system. This is achieved by invoking CMake in the following way:

```
cmake                                                    \
    -DCMAKE_TOOLCHAIN_FILE=/path/to/hpx/cmake/toolchains/XeonPhi.cmake \
    -DBOOST_ROOT=$BOOST_ROOT                             \
    -DHWLOC_ROOT=$HWLOC_ROOT                             \
    /path/to/hpx
```

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

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[125] page.

Note: This section of the manual is based off of our collaborators Patrick Diehl's blog post Installing HPX on Fedora 22[126].

- 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 ..
```

125 https://stellar-group.org/downloads/
126 http://diehlpk.github.io/2015/08/04/hpx-fedora.html

- 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[127] page.

- Install all packages for a minimal installation:

```
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 same as provided with 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
```

- For building 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 provided with 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*

[127] https://stellar-group.org/downloads/

- *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
#
# 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
#
# 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"
)
# 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
set(CMAKE_EXE_LINKER_FLAGS_INIT "-L/gpfs/bbp.cscs.ch/apps/bgas/tools/gcc/gcc-4.8.2/
↪install/lib64 -latomic -lrt" 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
set(HPX_WITH_PARCELPORT_IBVERBS ON 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")
#
# 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
#
# 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)
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> " 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_IBVERBS_PARCELPORT OFF)
# Always disable the tcp parcelport as it is non-functional on the BGQ.
set(HPX_WITH_TCP_PARCELPORT OFF)
# Always enable the tcp parcelport as it is currently the only way to communicate on the BGQ.
set(HPX_WITH_MPI_PARCELPORT 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
#
# 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)
if (CMAKE_VERSION VERSION_GREATER 3.3.9)
  set(__includes "<INCLUDES>")
endif()
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 "<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 "")
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")
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 threshhold 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
#
# 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)
if (CMAKE_VERSION VERSION_GREATER 3.3.9)
  set(__includes "<INCLUDES>")
endif()
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 "<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 "")
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")
set(HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD "4096" CACHE STRING
  "The threshhold 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
#
# 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)
if (CMAKE_VERSION VERSION_GREATER 3.3.9)
  set(__includes "<INCLUDES>")
endif()
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 "")
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 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 threshhold 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
#
# 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)
if (CMAKE_VERSION VERSION_GREATER 3.3.9)
  set(__includes "<INCLUDES>")
endif()
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 "")
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
  "The threshhold in bytes to when perform zero copy optimizations (default: 128)")
```

XeonPhi

```
# Copyright (c) 2014 Thomas Heller
#
# 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_IBVERBS OFF CACHE BOOL "Enable the ibverbs based parcelport.␣
↪This is currently an experimental feature")
# 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*

Generic options

- *HPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT: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_CXX14_RETURN_TYPE_DEDUCTION:BOOL*
- *HPX_WITH_DATAPAR_BOOST_SIMD: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_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_HCC:BOOL*
- *HPX_WITH_HIDDEN_VISIBILITY:BOOL*
- *HPX_WITH_INCLUSIVE_SCAN_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_QUEUE_COMPATIBILITY:BOOL*
- *HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL*
- *HPX_WITH_SCOPED_UNLOCK_COMPATIBILITY:BOOL*
- *HPX_WITH_STACKOVERFLOW_DETECTION:BOOL*
- *HPX_WITH_STATIC_LINKING:BOOL*
- *HPX_WITH_SYCL:BOOL*
- *HPX_WITH_THREAD_COMPATIBILITY:BOOL*
- *HPX_WITH_UNWRAPPED_COMPATIBILITY:BOOL*
- *HPX_WITH_VIM_YCM:BOOL*
- *HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING*

HPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT:BOOL
Enable weaker (non-conforming) iterator requirements for parallel algorithms (default: OFF)

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 CUDA support (default: OFF)

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

HPX_WITH_CXX14_RETURN_TYPE_DEDUCTION:BOOL
Enable the use of auto as a return value in some places. Overriding this flag is only necessary if the C++ compiler is not standard compliant, e.g. nvcc.

HPX_WITH_DATAPAR_BOOST_SIMD:BOOL
Enable data parallel algorithm support using the external Boost.SIMD library (default: OFF)

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 only, default: ON)

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

HPX_WITH_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_HCC:BOOL
Enable hcc support (default: OFF)

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

HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY:BOOL
Enable old overloads for inclusive_scan (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, tbbmalloc, and custom (default is: tcmalloc)

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

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

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

HPX_WITH_QUEUE_COMPATIBILITY:BOOL
Enable old style queue components in API (default: OFF)

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

HPX_WITH_SCOPED_UNLOCK_COMPATIBILITY:BOOL
Enable backwards compatibility for scoped_unlock utility (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_SYCL:BOOL
Enable sycl support (default: OFF)

HPX_WITH_THREAD_COMPATIBILITY:BOOL
Use a compatibility implementation of std::thread, i.e. fall back to Boost.Thread (default: OFF)

HPX_WITH_UNWRAPPED_COMPATIBILITY:BOOL
Enable the deprecated unwrapped function (default: ON)

HPX_WITH_VIM_YCM:BOOL
Generate HPX completion file for VIM YouCompleteMe plugin

HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING
The threshhold 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_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_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
Build HPX runtime (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_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_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
Maximum number of terminated threads collected before those are cleaned up (default: 100)

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 (default: 64)

HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING
HPX applications will not run on machines with more NUMA domains (default: 4)

HPX_WITH_MORE_THAN_64_THREADS:BOOL
HPX applications will be able to run on more than 64 cores (default: OFF)

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_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: 5)

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: ON)

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_MPI_ENV:STRING*
- *HPX_WITH_PARCELPORT_MPI_MULTITHREADED: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_MPI_ENV:STRING
List of environment variables checked to detect MPI (default: MV2_COMM_WORLD_RANK;PMI_RANK;OMPI_COMM_WO

HPX_WITH_PARCELPORT_MPI_MULTITHREADED:BOOL
Turn on MPI multithreading support (default: ON).

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_TESTS_DEBUG_LOG:BOOL*
- *HPX_WITH_TESTS_DEBUG_LOG_DESTINATION: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_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_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)

Additional tools and libraries used by *HPX*

Here is a list of additional libraries and tools which 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[128] installation to be used for compiling *HPX* Set this if CMake is not able to locate a suitable version of Boost[129] The directory specified here can be either the root of a installed Boost distribution or the directory where you unpacked and built Boost[130] without installing it (with staged libraries).

HWLOC_ROOT:PATH
Specifies where to look for the Portable Hardware Locality (HWLOC)[131] library. Set this if CMake is not able to locate a suitable version of Portable Hardware Locality (HWLOC)[132] Portable Hardware Locality (HWLOC)[133] 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 Performance Application Programming Interface (PAPI)[134] library. The PAPI library is necessary 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(tm) product, either Intel Amplifier(tm) or Intel Inspector(tm). 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.

[128] https://www.boost.org/
[129] https://www.boost.org/
[130] https://www.boost.org/
[131] https://www.open-mpi.org/projects/hwloc/
[132] https://www.open-mpi.org/projects/hwloc/
[133] https://www.open-mpi.org/projects/hwloc/
[134] https://icl.cs.utk.edu/papi/

```
//  Copyright (c) 2007-2012 Hartmut Kaiser
//
//  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.

//[simplest_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/include/iostreams.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 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).

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 which 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/include/iostreams.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

```
#if !defined(HELLO_WORLD_COMPONENT_HPP)
#define HELLO_WORLD_COMPONENT_HPP

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

#include <utility>

namespace examples { namespace server
{
    struct HPX_COMPONENT_EXPORT hello_world
        : hpx::components::component_base<hello_world>
    {
        void invoke();
        HPX_DEFINE_COMPONENT_ACTION(hello_world, invoke);
    };
}}

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action);
```

```
namespace examples
{
    struct hello_world
      : hpx::components::client_base<hello_world, server::hello_world>
    {
        typedef hpx::components::client_base<hello_world, server::hello_world>
            base_type;

        hello_world(hpx::future<hpx::naming::id_type> && f)
          : base_type(std::move(f))
        {}

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

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

#endif // HELLO_WORLD_COMPONENT_HPP
```

hello_world_client.cpp

```
//  Copyright (c) 2012 Bryce Lelbach
//
//  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)

//[hello_world_client_getting_started
#include "hello_world_component.hpp"
#include <hpx/hpx_init.hpp>

int hpx_main(boost::program_options::variables_map&)
{
    {
        // 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 hpx::finalize(); // Initiate shutdown of the runtime system.
}

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

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 your existing, or new CMakeLists.txt you can leverage the find_package[135] command integrated into CMake. Following is a Hello World component example using CMake.

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

- hello_world_component.hpp
- hello_world_component.cpp

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

- 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 3.3.2 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 MAKE_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.

The simplest way to add the *HPX* component is to use the add_hpx_component macro and add it to the CMakeLists.txt file:

```
# 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: Treat this component as a plugin-able library
- COMPILE_FLAGS: Additional compiler flags
- LINK_FLAGS: Additional linker flags
- FOLDER: Add 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[136] in your targets. The *HPX* include directories are available with the HPX_INCLUDE_DIRS CMake variable.

CMake macros to integrate *HPX* into existing applications

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 a install rule for the target
- PLUGIN: Treat 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

[136] https://www.cmake.org/cmake/help/latest/command/target_link_libraries.html

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 which 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[137].

Using HPX with Makefile

A basic project building with *HPX* is through creating makefiles. The process of creating one can get complex depending upon the use of cmake parameter `HPX_WITH_HPX_MAIN` (which defaults to ON).

How to build *HPX* applications with makefile

If *HPX* is installed correctly, you should be able to build and run a simple hello world program. It prints `Hello World!` on the *locality* you run it on.

```
//  Copyright (c) 2007-2012 Hartmut Kaiser
//
//  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.

//[simplest_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/include/iostreams.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
```

137 https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/RPATH-handling

```
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 successfull 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 which 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/include/iostreams.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

```
#if !defined(HELLO_WORLD_COMPONENT_HPP)
#define HELLO_WORLD_COMPONENT_HPP

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

#include <utility>

namespace examples { namespace server
{
    struct HPX_COMPONENT_EXPORT hello_world
        : hpx::components::component_base<hello_world>
    {
        void invoke();
        HPX_DEFINE_COMPONENT_ACTION(hello_world, invoke);
    };
}}

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action);

namespace examples
{
    struct hello_world
      : hpx::components::client_base<hello_world, server::hello_world>
    {
        typedef hpx::components::client_base<hello_world, server::hello_world>
            base_type;

        hello_world(hpx::future<hpx::naming::id_type> && f)
          : base_type(std::move(f))
        {}

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

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

```
#endif // HELLO_WORLD_COMPONENT_HPP
```

hello_world_client.cpp

```
//  Copyright (c) 2012 Bryce Lelbach
//
//  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)

//[hello_world_client_getting_started
#include "hello_world_component.hpp"
#include <hpx/hpx_init.hpp>

int hpx_main(boost::program_options::variables_map&)
{
    {
        // 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 hpx::finalize(); // Initiate shutdown of the runtime system.
}

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

Now in the directory, create a Makefile. Add the following code:

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

CXXFLAGS=-O3 -std=c++17

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

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

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

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

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

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

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

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

To build the program, type:

```
make
```

A successfull 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 and provides with 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. All *HPX* API functions can be used from within this function now.

Note: The function `main()` does not need to expect receiving `argc 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 Mac OSX, 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 Mac OSX. 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(boost::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.

Important: You cannot use any of the *HPX* API functions other that *hpx::stop* from inside your main() function.

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(boost::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[138]. This is a structured text format allowing to group key/value pairs (properties) into sections. The basic element contained in an ini file is the property. Every property has a name and a value, delimited by an equals sign `'='`. The name appears to the left of the equals sign:

```
name=value
```

The value may contain equal signs as only the first `'='` character is interpreted as the delimiter between `name` and `value` Whitespace before the name, after the value and immediately before and after the delimiting equal sign is ignored. Whitespace inside the value is retained.

Properties may be grouped into arbitrarily named sections. The section name appears on a line by itself, in square brackets `[` and `]`. All properties after the section declaration are associated with that section. There is no explicit "end of section" delimiter; sections end at the next section declaration, or the end of the file:

```
[section]
```

In *HPX* sections can be nested. A nested section has a name composed of all section names it is embedded in. The section names are concatenated using a dot `'.'`:

```
[outer_section.inner_section]
```

Here `inner_section` is logically nested within `outer_section`.

It is possible to use the full section name concatenated with the property name to refer to a particular property. For example in:

```
[a.b.c]
d = e
```

the property value of `d` can be referred to as `a.b.c.d=e`.

In *HPX* ini files can contain comments. Hash signs `'#'` at the beginning of a line indicate a comment. All characters starting with the `'#'` until the end of line are ignored.

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

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>}

[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.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.medium_size	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`).
hpx.stacks.large_size	This is initialized to the large stack size to be used by *HPX*-threads. Set by default to the value of the compile time preprocessor constant `HPX_LARGE_STACK_SIZE` (defaults to `0x200000`). This setting is used by default for the thread running `hpx_main` only.
hpx.	This is initialized to the huge stack size to be used by *HPX*-threads. Set by default to the value of

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 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]}
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]}
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}
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 succesful, 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"

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 have 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[139] 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.

HPX Command Line Options

The predefined command line options for any application using `hpx::init` are described in the following subsections.

[139] The phase of a *HPX*-thread counts how often this thread has been activated.

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: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 configuraton 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
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 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 -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 -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.

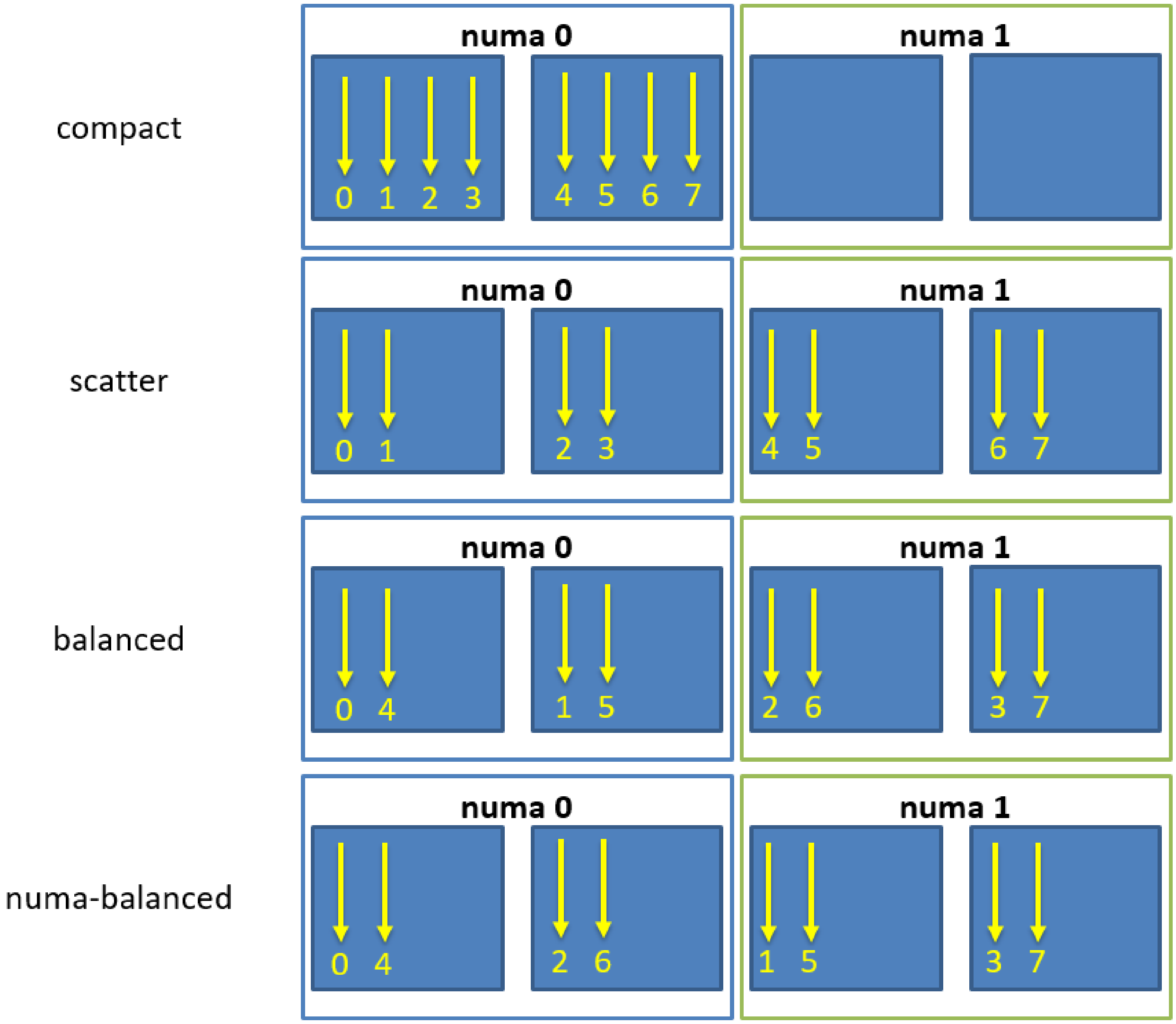

Fig. 2.7: Schematic of thread affinity type distributions.

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, in *HPX* we implement 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 provide synchronization for HPX applications. Most of them are familiar from other frameworks, but a few of them work in slightly special different ways adapted to 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;
c.set(42);
cout << c.get();      // will print '42'
```

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)
{
    cout << c.get();        // prints 42
    done.set();             // signal back
}

{
    hpx::lcos::local::channel<int> c;
    hpx::lcos::local::channel<> done;

    hpx::apply(&do_something, c, done);

    c.set(42);              // send some value
    done.get();             // wait for thread to be done
}
```

A channel component is created on one *locality* and can be send 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 some_action(hpx::lcos::channel<double> c)
{
    for (double d : c)
        hpx::cout << d << std::endl;
}
HPX_REGISTER_ACTION(some_action);

{
```

```
    // 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(some_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.

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 you must handle exceptions and it may cause a blocking call. Unwrapping can allow us 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 which 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[140], *HPX* adds functions allowing to compose several futures in a more flexible way.

[140] 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[141], `..._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[142], `..._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 we currently see 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[143] (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411[144] (Task Blocks), and N4406[145] (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 which 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

141 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
142 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
143 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf
144 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
145 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 indeterminately sequenced within each thread.

Important: It is the caller's responsibility to ensure correctness, for example 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 are 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/include/parallel_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/include/parallel_adjacent_find.hpp>	adjacent_find[146]
hpx::parallel::v1::all_of	Checks if a predicate is `true` for all of the elements in a range.	<hpx/include/parallel_all_any_none.hpp>	all_any_none_of[147]
hpx::parallel::v1::any_of	Checks if a predicate is `true` for any of the elements in a range.	<hpx/include/parallel_all_any_none.hpp>	all_any_none_of[148]
hpx::parallel::v1::count	Returns the number of elements equal to a given value.	<hpx/include/parallel_count.hpp>	count[149]
hpx::parallel::v1::count_if	Returns the number of elements satisfying a specific criteria.	<hpx/include/parallel_count.hpp>	count_if[150]
hpx::parallel::v1::equal	Determines if two sets of elements are the same.	<hpx/include/parallel_equal.hpp>	equal[151]
hpx::parallel::v1::exclusive_scan	Does an exclusive parallel scan over a range of elements.	<hpx/include/parallel_scan.hpp>	exclusive_scan[152]
hpx::parallel::v1::find	Finds the first element equal to a given value.	<hpx/include/parallel_find.hpp>	find[153]
hpx::parallel::v1::find_end	Finds the last sequence of elements in a certain range.	<hpx/include/parallel_find.hpp>	find_end[154]
hpx::parallel::v1::find_first_of	Searches for any one of a set of elements.	<hpx/include/parallel_find.hpp>	find_first_of[155]
hpx::parallel::v1::find_if	Finds the first element satisfying a specific criteria.	<hpx/include/parallel_find.hpp>	find[156]
hpx::parallel::v1::find_if_not	Finds the first element not satisfying a specific criteria.	<hpx/include/parallel_find.hpp>	find_if_not[157]
hpx::parallel::v1::for_each	Applies a function to a range of elements.	<hpx/include/parallel_for_each.hpp>	for_each[158]
hpx::parallel::v1::for_each_n	Applies a function to a number of elements.	<hpx/include/parallel_for_each.hpp>	for_each_n[159]
hpx::parallel::v1::inclusive_scan	Does an inclusive parallel scan over a range of elements.	<hpx/include/parallel_scan.hpp>	inclusive_scan[160]
hpx::parallel::v1::lexicographical_compare	Checks if a range of values is lexicographically less than another range of values.	<hpx/include/parallel_lexicographical_compare.hpp>	lexicographical_compare[161]
hpx::parallel::v1::mismatch	Finds the first position where two ranges differ.	<hpx/include/parallel_mismatch.hpp>	mismatch[162]
hpx::parallel::v1::none_of	Checks if a predicate is `true` for none of the elements in a range.	<hpx/include/parallel_all_any_none.hpp>	all_any_none_of[163]
hpx::parallel::v1::search	Searches for a range of elements.	<hpx/include/parallel_search.hpp>	search[164]
hpx::parallel::v1::search_n	Searches for a number consecutive copies of an element in a range.	<hpx/include/parallel_search.hpp>	search_n[165]

[146] http://en.cppreference.com/w/cpp/algorithm/adjacent_find
[147] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[148] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[149] http://en.cppreference.com/w/cpp/algorithm/count
[150] http://en.cppreference.com/w/cpp/algorithm/count_if
[151] http://en.cppreference.com/w/cpp/algorithm/equal
[152] http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
[153] http://en.cppreference.com/w/cpp/algorithm/find
[154] http://en.cppreference.com/w/cpp/algorithm/find_end
[155] http://en.cppreference.com/w/cpp/algorithm/find_first_of
[156] http://en.cppreference.com/w/cpp/algorithm/find
[157] http://en.cppreference.com/w/cpp/algorithm/find_if_not
[158] http://en.cppreference.com/w/cpp/algorithm/for_each
[159] http://en.cppreference.com/w/cpp/algorithm/for_each_n
[160] http://en.cppreference.com/w/cpp/algorithm/inclusive_scan
[161] http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
[162] http://en.cppreference.com/w/cpp/algorithm/mismatch
[163] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[164] http://en.cppreference.com/w/cpp/algorithm/search
[165] http://en.cppreference.com/w/cpp/algorithm/search_n

Table 2.16: Modifying Parallel Algorithms (In Header: <*hpx/include/parallel_algorithm.hpp*>)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::copy	Copies a range of elements to a new location.	`<hpx/include/parallel_copy.hpp>`	exclusive_scan[166]
hpx::parallel::v1::copy_n	Copies a number of elements to a new location.	`<hpx/include/parallel_copy.hpp>`	copy_n[167]
hpx::parallel::v1::copy_if	Copies the elements from a range to a new location for which the given predicate is `true`	`<hpx/include/parallel_copy.hpp>`	copy[168]
hpx::parallel::v1::move	Moves a range of elements to a new location.	`<hpx/include/parallel_fill.hpp>`	move[169]
hpx::parallel::v1::fill	Assigns a range of elements a certain value.	`<hpx/include/parallel_fill.hpp>`	fill[170]
hpx::parallel::v1::fill_n	Assigns a value to a number of elements.	`<hpx/include/parallel_fill.hpp>`	fill_n[171]
hpx::parallel::v1::generate	Saves the result of a function in a range.	`<hpx/include/parallel_generate.hpp>`	generate[172]
hpx::parallel::v1::generate_n	Saves the result of N applications of a function.	`<hpx/include/parallel_generate.hpp>`	generate_n[173]
hpx::parallel::v1::remove	Removes the elements from a range that are equal to the given value.	`<hpx/include/parallel_remove.hpp>`	remove[174]
hpx::parallel::v1::remove_if	Removes the elements from a range that are equal to the given predicate is `false`	`<hpx/include/parallel_remove.hpp>`	remove[175]
hpx::parallel::v1::remove_copy	Copies the elements from a range to a new location that are not equal to the given value.	`<hpx/include/parallel_remove_copy.hpp>`	remove_copy[176]
hpx::parallel::v1::remove_copy_if	Copies the elements from a range to a new location for which the given predicate is `false`	`<hpx/include/parallel_remove_copy.hpp>`	remove_copy[177]
hpx::parallel::v1::replace	Replaces all values satisfying specific criteria with another value.	`<hpx/include/parallel_replace.hpp>`	replace[178]
hpx::parallel::v1::replace_if	Replaces all values satisfying specific criteria with another value.	`<hpx/include/parallel_replace.hpp>`	replace[179]
hpx::parallel::v1::replace_copy	Copies a range, replacing elements satisfying specific criteria with another value.	`<hpx/include/parallel_replace.hpp>`	replace_copy[180]
hpx::parallel::v1::replace_copy_if	Copies a range, replacing elements satisfying specific criteria with another value.	`<hpx/include/parallel_replace.hpp>`	replace_copy[181]
hpx::parallel::v1::reverse	Reverses the order elements in a range.	`<hpx/include/parallel_reverse.hpp>`	reverse[182]
hpx::parallel::v1::reverse_copy	Creates a copy of a range that is reversed.	`<hpx/include/parallel_reverse.hpp>`	reverse_copy[183]

Table 2.17: Set operations on sorted sequences (In Header: *<hpx/include/parallel_algorithm.hpp>*)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::merge	Merges two sorted ranges.	<hpx/include/parallel_merge.hpp>	merge[190]
hpx::parallel::v1::inplace_merge	Merges two ordered ranges in-place.	<hpx/include/parallel_merge.hpp>	inplace_merge[191]
hpx::parallel::v1::includes	Returns true if one set is a subset of another.	<hpx/include/parallel_set_operations.hpp>	includes[192]
hpx::parallel::v1::set_difference	Computes the difference between two sets.	<hpx/include/parallel_set_operations.hpp>	set_difference[193]
hpx::parallel::v1::set_intersection	Computes the intersection of two sets.	<hpx/include/parallel_set_operations.hpp>	set_intersection[194]
hpx::parallel::v1::set_symmetric_difference	Computes the symmetric difference between two sets.	<hpx/include/parallel_set_operations.hpp>	set_symmetric_difference[195]
hpx::parallel::v1::set_union	Computes the union of two sets.	<hpx/include/parallel_set_operations.hpp>	set_union[196]

[166] http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
[167] http://en.cppreference.com/w/cpp/algorithm/copy_n
[168] http://en.cppreference.com/w/cpp/algorithm/copy
[169] http://en.cppreference.com/w/cpp/algorithm/move
[170] http://en.cppreference.com/w/cpp/algorithm/fill
[171] http://en.cppreference.com/w/cpp/algorithm/fill_n
[172] http://en.cppreference.com/w/cpp/algorithm/generate
[173] http://en.cppreference.com/w/cpp/algorithm/generate_n
[174] http://en.cppreference.com/w/cpp/algorithm/remove
[175] http://en.cppreference.com/w/cpp/algorithm/remove
[176] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[177] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[178] http://en.cppreference.com/w/cpp/algorithm/replace
[179] http://en.cppreference.com/w/cpp/algorithm/replace
[180] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[181] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[182] http://en.cppreference.com/w/cpp/algorithm/reverse
[183] http://en.cppreference.com/w/cpp/algorithm/reverse_copy
[184] http://en.cppreference.com/w/cpp/algorithm/rotate
[185] http://en.cppreference.com/w/cpp/algorithm/rotate_copy
[186] http://en.cppreference.com/w/cpp/algorithm/swap_ranges
[187] http://en.cppreference.com/w/cpp/algorithm/transform
[188] http://en.cppreference.com/w/cpp/algorithm/unique
[189] http://en.cppreference.com/w/cpp/algorithm/unique_copy
[190] http://en.cppreference.com/w/cpp/algorithm/merge
[191] http://en.cppreference.com/w/cpp/algorithm/inplace_merge
[192] http://en.cppreference.com/w/cpp/algorithm/includes
[193] http://en.cppreference.com/w/cpp/algorithm/set_difference
[194] http://en.cppreference.com/w/cpp/algorithm/set_intersection
[195] http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
[196] http://en.cppreference.com/w/cpp/algorithm/set_union

Table 2.18: Heap operations (In Header: <hpx/include/parallel_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/include/is_heap.hpp>`	is_heap[197]
`hpx::parallel::v1::is_heap_until`	Returns the first element that breaks a max heap.	`<hpx/include/is_heap.hpp>`	is_heap_until[198]

Table 2.19: Minimum/maximum operations (In Header: <hpx/include/parallel_algortithm.hpp>)

Name	Description	In header	Algorithm page at cppreference.com
`hpx::parallel::v1::max_element`	Returns the largest element in a range.	`<hpx/include/parallel_minmax.hpp>`	max_element[199]
`hpx::parallel::v1::min_element`	Returns the smallest element in a range.	`<hpx/include/parallel_minmax.hpp>`	min_element[200]
`hpx::parallel::v1::minmax_element`	Returns the smallest and the largest element in a range.	`<hpx/include/parallel_minmax.hpp>`	minmax_element[201]

Table 2.20: Partitioning Operations (In Header: *<hpx/include/parallel_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/include/parallel_is_partitioned.hpp>`	is_partitioned[202]
`hpx::parallel::v1::partition`	Divides elements into two groups while don't preserve their relative order	`<hpx/include/parallel_partition.hpp>`	partition[203]
`hpx::parallel::v1::partition_copy`	Copies a range dividing the elements into two groups	`<hpx/include/parallel_partition.hpp>`	partition_copy[204]
`hpx::parallel::v1::stable_partition`	Divides elements into two groups while preserving their relative order	`<hpx/include/parallel_partition.hpp>`	stable_partition[205]

[197] http://en.cppreference.com/w/cpp/algorithm/is_heap
[198] http://en.cppreference.com/w/cpp/algorithm/is_heap_until
[199] http://en.cppreference.com/w/cpp/algorithm/max_element
[200] http://en.cppreference.com/w/cpp/algorithm/min_element
[201] http://en.cppreference.com/w/cpp/algorithm/minmax_element
[202] http://en.cppreference.com/w/cpp/algorithm/is_partitioned
[203] http://en.cppreference.com/w/cpp/algorithm/partition
[204] http://en.cppreference.com/w/cpp/algorithm/partition_copy
[205] http://en.cppreference.com/w/cpp/algorithm/stable_partition

Table 2.21: Sorting Operations (In Header: <*hpx/include/parallel_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/include/parallel_is_sorted.hpp>`	is_sorted[206]
`hpx::parallel::v1::is_sorted_until`	Returns the first unsorted element	`<hpx/include/parallel_is_sorted.hpp>`	is_sorted_until[207]
`hpx::parallel::v1::sort`	Sorts the elements in a range	`<hpx/include/parallel_sort.hpp>`	sort[208]
`hpx::parallel::v1::sort_by_key`	Sorts one range of data using keys supplied in another range	`<hpx/include/parallel_sort.hpp>`	

Table 2.22: Numeric Parallel Algorithms (In Header: <*hpx/include/parallel_numeric.hpp*>)

Name	Description	In header	Algorithm page at cppreference.com
`hpx::parallel::v1::adjacent_difference`	Calculates the difference between each element in an input range and the preceding element.	`<hpx/include/parallel_adjacent_difference.hpp>`	adjacent_difference[209]
`hpx::parallel::v1::reduce`	Sums up a range of elements.	`<hpx/include/parallel_reduce.hpp>`	reduce[210]
`hpx::parallel::v1::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/include/parallel_reduce.hpp>`	
`hpx::parallel::v1::transform_reduce`	Sums up a range of elements after applying a function. Also, accumulates the inner products of two input ranges.	`<hpx/include/parallel_transform_reduce.hpp>`	transform_reduce[211]
`hpx::parallel::v1::transform_inclusive_scan`	Does an inclusive parallel scan over a range of elements after applying a function.	`<hpx/include/parallel_scan.hpp>`	transform_inclusive_scan[212]
`hpx::parallel::v1::transform_exclusive_scan`	Does an exclusive parallel scan over a range of elements after applying a function.	`<hpx/include/parallel_scan.hpp>`	transform_exclusive_scan[213]

206 http://en.cppreference.com/w/cpp/algorithm/is_sorted
207 http://en.cppreference.com/w/cpp/algorithm/is_sorted_until
208 http://en.cppreference.com/w/cpp/algorithm/sort

Table 2.23: Dynamic Memory Management (In Header: <*hpx/include/parallel_memory.hpp*>)

Name	Description	In header	Algorithm page at cppreference.com
`hpx::parallel::v1::destroy`	Destroys a range of objects.	`<hpx/include/parallel_destroy.hpp>`	destroy[214]
`hpx::parallel::v1::destroy_n`	Destroys a range of objects.	`<hpx/include/parallel_destroy.hpp>`	destroy_n[215]
`hpx::parallel::v1::uninitialized_copy`	Copies a range of objects to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_copy.hpp>`	uninitialized_copy[216]
`hpx::parallel::v1::uninitialized_copy_n`	Copies a number of objects to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_copy.hpp>`	uninitialized_copy_n[217]
`hpx::parallel::v1::uninitialized_default_construct`	Copies a range of objects to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_default_construct.hpp>`	uninitialized_default_construct[218]
`hpx::parallel::v1::uninitialized_default_construct_n`	Copies a number of objects to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_default_construct.hpp>`	uninitialized_default_construct_n[219]
`hpx::parallel::v1::uninitialized_fill`	Copies an object to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_fill.hpp>`	uninitialized_fill[220]
`hpx::parallel::v1::uninitialized_fill_n`	Copies an object to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_fill.hpp>`	uninitialized_fill_n[221]
`hpx::parallel::v1::uninitialized_move`	Moves a range of objects to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_move.hpp>`	uninitialized_move[222]
`hpx::parallel::v1::uninitialized_move_n`	Moves a number of objects to an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_move.hpp>`	uninitialized_move_n[223]
`hpx::parallel::v1::uninitialized_value_construct`	Constructs objects in an uninitialized area of memory.	`<hpx/include/parallel_uninitialized_value_construct.hpp>`	uninitialized_value_construct[224]
`hpx::parallel::v1::uninitialized_value_construct_n`	Constructs objects in an uninitialized area of memory.	`<hpx/include/uninitialized_value_construct.hpp>`	uninitialized_value_construct_n[225]

[209] http://en.cppreference.com/w/cpp/algorithm/adjacent_difference
[210] http://en.cppreference.com/w/cpp/algorithm/reduce
[211] http://en.cppreference.com/w/cpp/algorithm/transform_reduce
[212] http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan
[213] http://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan

Table 2.24: Index-based for-loops (In Header: <*hpx/include/parallel_algorithm.hpp*>)

Name	Description	In header
hpx::parallel::v2::for_loop	Implements loop functionality over a range specified by integral or iterator bounds.	<hpx/include/parallel_for_loop.hpp>
hpx::parallel::v2::for_loop_strided	Implements loop functionality over a range specified by integral or iterator bounds.	<hpx/include/parallel_for_loop.hpp>
hpx::parallel::v2::for_loop_n	Implements loop functionality over a range specified by integral or iterator bounds.	<hpx/include/parallel_for_loop.hpp>
hpx::parallel::v2::for_loop_n_strided	Implements loop functionality over a range specified by integral or iterator bounds.	<hpx/include/parallel_for_loop.hpp>

Executor parameters and executor parameter traits

In *HPX* we introduce the notion of execution parameters and execution parameter traits. At this point, the only parameter which 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 (potential platform-specific) algorithms of determining those chunks 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 such a executor parameter type).

Other functions in the interface exist to discover whether a executor parameter type should be invoked once (i.e. returns a static chunk size, see *hpx::parallel::execution::static_chunk_size*) or whether it

[214] http://en.cppreference.com/w/cpp/memory/destroy
[215] http://en.cppreference.com/w/cpp/memory/destroy_n
[216] http://en.cppreference.com/w/cpp/memory/uninitialized_copy
[217] http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n
[218] http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct
[219] http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n
[220] http://en.cppreference.com/w/cpp/memory/uninitialized_fill
[221] http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n
[222] http://en.cppreference.com/w/cpp/memory/uninitialized_move
[223] http://en.cppreference.com/w/cpp/memory/uninitialized_move_n
[224] http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct
[225] http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct_n

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*).

Though this interface appears to require executor parameter type authors to implement all different basic operations, there is really none required. In practice, all operations have sensible defaults. However, some executor parameter types will naturally specialize all operations for maximum efficiency.

In *HPX* we have implemented 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 parameters 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 evenly (if possible) 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 an core finishes one chunk, it is dynamically assigned another If the size is not specified, the default chunk size is 1. This executor parameters 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. 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 parameters 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[226] are based on the `task_block` concept that is a part of the common subset of the Microsoft Parallel Patterns Library (PPL)[227] and the Intel Threading Building Blocks (TBB)[228] libraries.

This implementations adopts a simpler syntax than exposed by those libraries— one that is influenced by language-based concepts such as spawn and sync from Cilk++[229] and async and finish from X10[230]. It improves 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)
```

226 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
227 https://msdn.microsoft.com/en-us/library/dd492418.aspx
228 https://www.threadingbuildingblocks.org/
229 https://software.intel.com/en-us/articles/intel-cilk-plus/
230 https://x10-lang.org/

```
{
    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 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

In *HPX* we implemented some extensions for `task_block` beyond the actual standards proposal N4411[231]. The main addition is that a `task_block` can be invoked with a 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

[231] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf

of the invocation of a task block. Enabling passing an execution policy to `define_task_block` gives the user control over the amount of parallelism employed by the created `task_block`. In the following example the use of an explicit `par` execution policy makes the user's intent explicit:

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

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

    return compute(n) + left + right;
}
```

This also causes the *`hpx::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 which was used to create a given `task_block`.

Using executors to run tasks

Often, we want to be able to not only define an execution policy to use by default for all spawned tasks inside the task block, but in addition 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 which 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. 2.8 below shows an overview of the main API exposed by HPX. This shows the function invocation syntax as defined by the C++ language (dark gray), the additional invocation syntax as provided through C++ Standard Library features (medium gray), and the extensions added by *HPX* (light gray) where:

- `f` function to invoke,
- `p..`: (optional) arguments,
- `R`: return type of `f`,
- `action`: action type defined by, *`HPX_DEFINE_PLAIN_ACTION`* or *`HPX_DEFINE_COMPONENT_ACTION`* encapsulating `f`,
- `a`: an instance of the type `` ` ``action``,
- `id`: the global address the action is applied to.

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 mean time, 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()),
```

```
        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
// 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 );

// 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[232]

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[233]

232 http://en.cppreference.com/w/cpp/container/vector
233 http://en.cppreference.com/w/cpp/container/unordered_map

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)
    {
        *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/lcos/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/lcos/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;
```

```
    // Instanciate 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:

```cpp
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

using Vec = hpx::partitioned_vector<double>;
using View_2D = hpx::partitioned_vector_view<double,2>;

/* Do some code */

Vec v;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t height, width;

    // Instanciate the view
    View_2D vv(block, v.begin(), v.end(), {height,width});

    // The l-value is a view subscript, the image that owns vv(1,0)
    // evaluates the assignment.
    vv(0,1) = vv(1,0);

    // The l-value is a view subscript, the image that owns the r-value
    // (result of expression 'std::vector<double>(4,1.0)') evaluates the
    // assignment : oops! race between all participating images.
    vv(2,3) = std::vector<double>(4,1.0);
}
```

Iterator-based operations

Here are some examples of using iterators in the 3-D view case:

```cpp
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(int);

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;

    // Instanciate 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>;
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;

    // Instanciate the view
    View_1D vv(block, v.begin(), v.end(), {num_segments});

    // Instanciate the local view from the view
```

```
    auto local_vv = hpx::local_view(vv);

    for ( auto i = localvv.begin(); i != localvv.end(); i++ )
    {
        std::vector<float> & segment = *i;

        /* Do some code */
    }

}
```

Instanciating 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;

    // Instanciate the view
    View_2D vv(block, v.begin(), v.end(), {N,N});

    // Instanciate the subview
    View_2D svv(
        block,&vv(tilesize,0),&vv(2*tilesize-1,tilesize-1),{tilesize,tilesize},{N,N});

    if(block.this_image() == 0)
    {
        // Equivalent to 'vv(tilesize,0) = 2.0f'
        svv(0,0) = 2.0f;

        // Equivalent to 'vv(2*tilesize-1,tilesize-1) = 3.0f'
        svv(tilesize-1,tilesize-1) = 3.0f;
    }

}
```

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 instanciate this view. We illustrate below how a single constructor call can perform those two operations:

```
#include <hpx/components/containers/coarray/coarray.hpp>
#include <hpx/lcos/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/lcos/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++)
    {
        // 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 out of the box 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[234] for more information), below is a minimal example to illustrate the approach. As a test application we will use the multithreaded `hello_world` 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
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 those have 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
> ```

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
APP_OPTIONS=

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

[234] http://www.clusterresources.com/torquedocs21/

`APP_PATH` and `APP_OPTIONS` are shell variables that respectively specify the correct path to the executable (`hello_world` in this case) and the command line options. Since the `hello_world` 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. We 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, we invoke this script using the pbsdsh tool:

```
#!/bin/bash
#
#PBS -l nodes=2:ppn=4

APP_PATH=~/packages/hpx/bin/hello_world
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 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 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 machines documentation to get a list of existing partitions) and `<number-of-nodes>` is the number of compute nodes you want to use. 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 which 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, we suggest 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 you can issue the following command:

```
srun -p <node-type> -N <number-of-nodes> --pty /bin/bash -l
```

After the shell has been opened, you can run your 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 you 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 you need to add `#SBATCH` comments in your script or provide the necessary parameters to `sbatc` 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 code 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 you 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 you may end up with false positives from *HPX* itself when using sanitizers for your application.

To use sanitizers with *HPX* you should turn on `HPX_WITH_SANITIZERS` and turn off `HPX_WITH_STACK_OVERFLOW_DETECTION` during CMake configuration. It's recommended to also build Boost with the same sanitizers that you will be using 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 of 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 which expose a predefined interface. *HPX* exposes special API functions that allow one to create, manage, 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 this 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 simple examples

An instance for 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
```

At application start, *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 later 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 its instance name.

Not all simple counter types require specifying all 4 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 2 types of wildcard characters:

- Wildcard characters in the performance counter type
- Wildcard characters in the performance counter instance name

Wildcard character have 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 or via 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 which uses `hpx::init` for its initialization. While there are much more command line options available (see *HPX Command Line Options*), the set of options related to Performance Counters allow one to list existing counters, 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`	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.
`--hpx:list-counter-infos`	list the description of all registered performance counters.
`--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)

While the options `--hpx:list-counters` and `--hpx:list-counter-infos` give a short listing 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 for instance tested using the `hello_world` example.

Listing all available counters `hello_world --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 --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` with a list of counter names:

```
hello_world \
    --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 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 6 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.

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 \
    --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` 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 4 worker threads for the execution of HPX threads (see command line option *`--hpx:threads`*) the following command line

```
hello_world \
    --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 header.

With format as csv:

```
hello_world \
    --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 full countername as 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 \
    --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 short countername as 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 \
    --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` to be used with `--hpx:print-counter-format` to print performance counter values in CSV format without any header:

```
hello_world \
--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 allowing 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 always not more than one active instance of any of the existing performance counters.

In order to access the counter value (or invoking 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 [classref 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 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, e.g. 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 to instantiate exactly one counter instance for each *locality*. Repeated requests to instantiate such a counter will return the same instance, e.g. 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 [hpx_link examples/performance_counters/simplest_performance_counter.cpp..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) 2007-2018 Hartmut Kaiser
//
//  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)

#if !defined(HPX_PERFORMANCE_COUNTERS_PERFORMANCE_COUNTER_JAN_18_2013_0939AM)
#define HPX_PERFORMANCE_COUNTERS_PERFORMANCE_COUNTER_JAN_18_2013_0939AM

#include <hpx/config.hpp>
#include <hpx/lcos/future.hpp>
#include <hpx/runtime/components/client_base.hpp>
#include <hpx/runtime/launch_policy.hpp>
#include <hpx/util/bind_front.hpp>

#include <hpx/performance_counters/counters_fwd.hpp>
#include <hpx/performance_counters/stubs/performance_counter.hpp>

#include <string>
#include <utility>
#include <vector>

///////////////////////////////////////////////////////////////////////////////
namespace hpx { namespace performance_counters
{
    ///////////////////////////////////////////////////////////////////////////
    struct HPX_EXPORT performance_counter
      : components::client_base<performance_counter, stubs::performance_counter>
    {
```

```
    typedef components::client_base<
        performance_counter, stubs::performance_counter
    > base_type;

    performance_counter() {}

    performance_counter(std::string const& name);

    performance_counter(std::string const& name, hpx::id_type const& locality);

    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();
    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 wildcards).
    HPX_API_EXPORT std::vector<performance_counter> discover_counters(
        std::string const& name, error_code& ec = throws);
}}

#endif
```

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
//
//  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)

#if !defined(HPX_PERFORMANCE_COUNTERS_BASE_PERFORMANCE_COUNTER_JAN_18_2013_1036AM)
```

```
#define HPX_PERFORMANCE_COUNTERS_BASE_PERFORMANCE_COUNTER_JAN_18_2013_1036AM

#include <hpx/config.hpp>
#include <hpx/performance_counters/counters.hpp>
#include <hpx/performance_counters/server/base_performance_counter.hpp>
#include <hpx/runtime/actions/component_action.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 { 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();
        }
    };
}}

#endif
```

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
//
//  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)

#if !defined(PERFORMANCE_COUNTERS_SINE_SEP_20_2011_0112PM)
#define PERFORMANCE_COUNTERS_SINE_SEP_20_2011_0112PM

#include <hpx/hpx.hpp>
#include <hpx/util/interval_timer.hpp>
#include <hpx/lcos/local/spinlock.hpp>
#include <hpx/performance_counters/base_performance_counter.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) {}
        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_;

        hpx::util::interval_timer timer_;
    };
}}}

#endif
```

i.e. the type `sine_counter` is derived from the base class passing the type as a template argument (please see [hpx_link examples/performance_counters/sine/server/sine.hpp..sine.hpp] 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[235]. This base class itself is derived from the

[235] http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

`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.

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
		None	Returns the overall total number of invocations of all *AGAS* services provided by the given *AGAS* service category since its creation.

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_ACTI was specified, this counter allows 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_ACTI was specified, this counter allows to

[236] A message can potentially consist of more than one *parcel*.

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
`/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	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	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**	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 `--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). Note that this counter will be available only for Boost V1.56 and newer.	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	Returns the current (statistically estimated) median value calculated	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

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 -t2 \
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
```

```
    --hpx:print-counter=/arithmetics/add@/threads{locality#0/worker-thread#*}/count/
↪cumulative
hello world from OS-thread 0 on locality 0
hello world from OS-thread 1 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.515640,[s],25
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.515520,[s],36
/arithmetics/add@/threads{locality#0/worker-thread#*}/count/cumulative,1,0.516445,[s],
↪64
```

Since all wildcards in the parameters are expanded, this example is fully equivalent to specifying both counters separately to `/arithmetics/add`:

```
./bin/hello_world -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 those actions, which 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[237], 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[238] configuration. TAU[239] is an optional dependency when using APEX[240].

To build *HPX* with APEX[241] add `HPX_WITH_APEX=ON`, and, optionally, `TAU_ROOT=$PATH_TO_TAU` to your CMake[242] configuration. In addition, you can override the tag used for APEX[243] with the `HPX_WITH_APEX_TAG` option. Please see the APEX *HPX* documentation[244] for detailed instructions on using APEX[245] 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.

[237] https://khuck.github.io/xpress-apex/
[238] https://www.cmake.org
[239] https://www.cs.uoregon.edu/research/tau/home.php
[240] https://khuck.github.io/xpress-apex/
[241] https://khuck.github.io/xpress-apex/
[242] https://www.cmake.org
[243] https://khuck.github.io/xpress-apex/
[244] https://khuck.github.io/xpress-apex/usage/#hpx-louisiana-state-university
[245] https://khuck.github.io/xpress-apex/

This scheduler can be used with two underlying queuing policies (FIFO: first-in-first-out, and LIFO: last-in-first-out). The default is FIFO. In order to use the LIFO policy use the command line option `--hpx:queuing=local-priority-lifo`.

Static priority scheduling policy

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

The static scheduling policy maintains one queue per OS thread from which each OS thread pulls its tasks (user threads). Threads are distributed in a round robin fashion. There is no thread stealing in this policy.

Local scheduling policy

- invoke using: `--hpx:queuing=local` (or `-ql`)
- flag to turn on for build: `HPX_THREAD_SCHEDULERS=all` or `HPX_THREAD_SCHEDULERS=local`

The local scheduling policy maintains one queue per OS thread from which each OS thread pulls its tasks (user threads).

Static scheduling policy

- invoke using: `--hpx:queuing=static`
- flag to turn on for build: `HPX_THREAD_SCHEDULERS=all` or `HPX_THREAD_SCHEDULERS=static`

The static scheduling policy maintains one queue per OS thread from which each OS thread pulls its tasks (user threads). Threads are distributed in a round robin fashion. There is no thread stealing in this policy.

Priority ABP scheduling policy

- invoke using: `--hpx:queuing=abp-priority-fifo`
- flag to turn on for build: `HPX_THREAD_SCHEDULERS=all` or `HPX_THREAD_SCHEDULERS=abp-priority`

Priority ABP policy maintains a double ended lock free queue for each OS thread. By default the number of high priority queues is equal to the number of OS threads; the number of high priority queues can be specified on the command line using `--hpx:high-priority-threads`. High priority threads are executed by the first OS threads before any other work is executed. When a queue is empty work will be taken from high priority queues first. There is one low priority queue from which threads will be scheduled only when there is no other work. For this scheduling policy there is an option to turn on NUMA sensitivity using the command line option `--hpx:numa-sensitive`. When NUMA sensitivity is turned on work stealing is done from queues associated with the same NUMA domain first, only after that work is stolen from other NUMA domains.

This scheduler can be used with two underlying queuing policies (FIFO: first-in-first-out, and LIFO: last-in-first-out). In order to use the LIFO policy use the command line option `--hpx:queuing=abp-priority-lifo`.

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[246] 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[247] communication. MPI[248] 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 to schedule important tasks with high priority instead of using a separate thread pool.

Using the resource partitioner

In order to create custom thread pools the resource partitioner needs to be set up before *HPX* is initialized by creating an instance of `hpx::resource::partitioner`:

```
#include <hpx/hpx_init.hpp>
#include <hpx/runtime/resource/partitioner.hpp>

int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

int main(int argc, char** argv)
{
    hpx::resource::partitioner rp(argc, argv);
    hpx::init();
}
```

Note that we have to pass `argc` and `argv` to the resource partitioner to be able to parse thread binding options passed on the command line. You should pass the same arguments to the `hpx::resource::partitioner` constructor as you would to `hpx::init` or `hpx::start`. Running the above code will have the same effect as not initializing it at all, i.e. a default thread pool will be created with the type and number of threads specified on the command line.

The resource partitioner class is the interface to add thread pools to the *HPX* runtime and to assign resources to the thread pools.

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

Then, to add resources to the thread pool you can use the `hpx::resource::partitioner::add_resource` method. The resource partitioner exposes the hardware topology retrieved using Portable Hardware Locality (HWLOC)[249] and lets you iterate through the topology to add the wanted processing units to the thread pool. Be-

[246] https://en.wikipedia.org/wiki/Message_Passing_Interface
[247] https://en.wikipedia.org/wiki/Message_Passing_Interface
[248] https://en.wikipedia.org/wiki/Message_Passing_Interface
[249] https://www.open-mpi.org/projects/hwloc/

low 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/runtime/resource/partitioner.hpp>

#include <iostream>

int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

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();
}
```

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.

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 is 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 we assume that the function `raise_exception()` is executed by invoking the plain action `raise_exception_type`.

```
void raise_exception()
{
    HPX_THROW_EXCEPTION(hpx::no_success, "raise_exception", "simulated error");
}
HPX_PLAIN_ACTION(raise_exception, raise_exception_action);
```

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:

```
        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:"
                << hpx::diagnostic_information(e) << "\n";
        }
        hpx::cout << hpx::flush;
```

Note: The exception is transferred back to the invocation site even if it is executed on a different *locality*.

Additionally, this example demonstrates how an exception thrown by an (possibly remote) action can be handled. It shows the use of `hpx::diagnostic_information` which retrieves all available diagnostic information from the exception as a formatted string. This includes, for instance, the name of the source file and line number, the sequence number of the OS-thread and the *HPX*-thread id, the *locality* id and the stack backtrace of the point where the original exception was thrown.

Under certain circumstances it is desirable to output only some of the diagnostics, or to output those using different formatting. For this case, *HPX* exposes a set of lower level functions as demonstrated in the following code snippet:

```
        hpx::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";
            hpx::cout << "{env}: "         << hpx::get_error_env(e) << "\n";
        }
        hpx::cout << hpx::flush;
```

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:

```
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
    // the exception.
    hpx::cout << "diagnostic information:"
        << hpx::diagnostic_information(ec) << "\n";
}

hpx::cout << hpx::flush;
```

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:

```
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";
    }

    hpx::cout << hpx::flush;
```

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 which will hold the error code only. The following snippet demonstrates its use:

```
    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";
        }

        hpx::cout << hpx::flush;
```

All functions which 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 in *HPX* we have provided several utilities to users. The following section documents those facilies.

Checkpoint

A common need of users is to periodically backup an application. This practice provides resiliency and potential restart points in code. We have developed the concept of a `checkpoint` to support this use case.

Found in `hpx/util/checkpoint.hpp`, `checkpoints` are defined as objects which 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 we 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. Let us look a simple use case below:

```
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.

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);
```

Now that we can create `checkpoint` s we now must 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(
    f_archive.get(), 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 is needed to be stored in an object, such as a file, for later use. For these cases we have provided two solutions: stream operator overloads and access iterators.

We have created the two stream overloads `operator<<` and `operator>>` to stream data out of and into `checkpoint`. You can see 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 an 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 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 and 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);
        int 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);
```

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 simplest possible ‘Hello world’ program as included as an example with *HPX*:

```
//  Copyright (c) 2007-2012 Hartmut Kaiser
//
//  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.

//[simplest_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/include/iostreams.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(
    simplest_hello_world
    SOURCES simplest_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 those is explicitly used.

2.6 Additional material

- 2-day workshop held at CSCS in 2016
  - Recorded lectures[250]
  - Slides[251]
- Tutorials repository[252]
- STE||AR Group blog posts[253]

2.7 HPX Modules

HPX itself is organized into different sub-libraries. Those libraries can be seen as independent modules, with clear dependencies and no cycles.

The tool create_library_skeleton.py[254] can be used to generate a basic skeleton. The structure of this skeleton is as follows:

- ***<lib_name>/***
    - *Readme.md*
    - *CMakeLists.txt*
    - *cmake*
    - *docs/ - index.rst*

250 https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOmIJ0lFNjKze18
251 https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2016
252 https://github.com/STEllAR-GROUP/tutorials
253 http://stellar-group.org/blog/
254 https://github.com/STEllAR-GROUP/hpx/blob/master/libs/create_library_skeleton.py

- ***examples/***
  - *CMakeLists.txt*
- ***include/***
  - ***hpx/***
    - *<lib_name>*
- ***src/***
  - *CMakeLists.txt*
- ***tests/***
  - *CMakeLists.txt*
  - ***unit/***
    - *CMakeLists.txt*
  - ***regressions/***
    - *CMakeLists.txt*
  - ***performance/***
    - *CMakeLists.txt*

A Readme should be always included which explains the basic purpose of the library and a link to the generated documentation.

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 seperation. The cmake subdirectory may include additional cmake scripts needed to generate the respective build configurations.

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.

In order to consume any library defined here, all you have to do is use *target_link_libraries* to get the dependencies. This of course requires that the library to link against specified the appropriate target include directories and libraries.

2.8 API reference

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 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

auto_chunk_size()

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*)

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.

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

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.

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

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.

Public Types

using const_iterator = std::vector::const_iterator

Public Functions

checkpoint()

checkpoint(*checkpoint* **const** &*c*)

checkpoint(*checkpoint* &&*c*)

~checkpoint()

checkpoint(std::vector<char> **const** &*vec*)

checkpoint(std::vector<char> &&*vec*)

checkpoint &**operator=**(*checkpoint* **const** &*c*)

checkpoint &**operator=**(*checkpoint* &&*c*)

bool **operator==**(*checkpoint* **const** &*c*) **const**

bool **operator!=**(*checkpoint* **const** &*c*) **const**

const_iterator **begin**() **const**

const_iterator **end**() **const**

size_t **size**() **const**

Private Functions

template <**typename** Archive>
void **serialize**(Archive &*arch*, **const** unsigned int *version*)

Private Members

std::vector<char> **data**

Friends

friend hpx::util::checkpoint::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*)

Resurrect

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).

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.

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

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

const std::size_t **invalid_core_id** = std::size_t(-1)

Friends

friend hpx::resource::core::pu

friend hpx::resource::core::numa_domain

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

struct dynamic_chunk_size
#include <dynamic_chunk_size.hpp> Loop iterations are divided into pieces of size *chunk_size* and then dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another If *chunk_size* is not specified, the default chunk size is 1.

Note This executor parameters type is equivalent to OpenMP's DYNAMIC scheduling directive.

Public Functions

dynamic_chunk_size (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.

class 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

Inherits from error_code

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::error_code::exception

error_code **make_error_code**(std::exception_ptr **const** &*e*)

class exception

#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.

Inherits from system_error

Subclassed by *hpx::exception_list*, *hpx::parallel::v2::task_canceled_exception*

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*).

class exception_list

#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.

Inherits from *hpx::exception*

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 *hpx*::*lcos*::local::spinlock **mutex_type**

typedef std::list<std::exception_ptr> **exception_list_type**

Private Members

exception_list_type **exceptions_**

mutex_type **mtx_**

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

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.

struct invoke

#include <invoke.hpp>

Public Functions

template <typename F, typename… *Ts*>

HPX_HOST_DEVICE util::invoke_result<F, Ts...>::type hpx::util::functional::invoke::ope

template <typename R>

struct invoke_r

#include <invoke.hpp>

Public Functions

template <typename F, typename… *Ts*>

HPX_HOST_DEVICE R hpx::util::functional::invoke_r::operator()(F && f, Ts &&... vs) con

template <typename T>

struct is_async_execution_policy

#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.

Inherits from execution::detail::is_async_execution_policy< hpx::util::decay< T >::type >

template <typename T>
struct is_execution_policy

#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.

Inherits from execution::detail::is_execution_policy< hpx::util::decay< T >::type >

template <typename T>
struct is_parallel_execution_policy

#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.

Inherits from execution::detail::is_parallel_execution_policy< hpx::util::decay< T >::type >

template <typename T>
struct is_sequenced_execution_policy

#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.

Inherits from execution::detail::is_sequenced_execution_policy< hpx::util::decay< T >::type >

struct launch

#include <launch_policy.hpp> Launch policies for *hpx::async* etc.

Inherits from detail::policy_holder<>

Public Functions

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.

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

const std::size_t **invalid_numa_domain_id** = std::size_t(-1)

Friends

friend hpx::resource::numa_domain::pu

friend hpx::resource::numa_domain::core

struct parallel_execution_tag
#include <execution_fwd.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 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

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<Parameters…>::typ
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.

executor_type **const** &**executor**() **const**
: Return the associated executor object.

executor_parameters_type &**parameters**()
: Return the associated executor parameters object.

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::parallel_policy::hpx::serialization::access

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

parallel_policy_executor(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.

template <**typename** Executor, **typename** Parameters>

struct parallel_policy_shim

#include <execution_policy.hpp> The class *parallel_policy_shim* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

Inherits from *hpx::parallel::execution::parallel_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

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<Parameters_…>::
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.

Executor **const** &**executor**() **const**
Return the associated executor object.

Parameters &**parameters**()
Return the associated executor parameters object.

Parameters **const** &**parameters**() **const**
Return the associated executor parameters object.

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

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<Parameters…>::typ
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.

executor_type **const** &**executor**() **const**
Return the associated executor object.

executor_parameters_type &**parameters**()
Return the associated executor parameters object.

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::parallel_task_policy::hpx::serialization::access

template <typename *Executor*, typename *Parameters*>
struct parallel_task_policy_shim

#include <execution_policy.hpp> Extension: The class *parallel_task_policy_shim* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading based on combining a underlying parallel_task_policy and an executor and indicate that a parallel algorithm's execution may be parallelized.

Inherits from *hpx::parallel::execution::parallel_task_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

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<Parameters_…>::
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.

Executor **const** &**executor**() **const**
Return the associated executor object.

Parameters &**parameters**()
Return the associated executor parameters object.

Parameters **const** &**parameters**() **const**
Return the associated executor parameters object.

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.

executor_type **const** &**executor**() **const**
Return the associated executor object.

executor_parameters_type &**parameters**()
Return the associated executor parameters object.

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::parallel_unsequenced_policy::hpx::serialization::acce

class partitioner
#include <partitioner.hpp>

Public Functions

partitioner(*util*::function_nonser<int) boost::program_options::variables_map &vm
> **const** &*f*, boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char *******argv*, std::vector<std::string> *ini_config*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *runtime_mode* *mode* = runtime_mode_default

partitioner(*util*::function_nonser<int) int, char **
> **const** &*f*, int *argc*, char **argv*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default

partitioner(*util*::function_nonser<int) int, char **
> **const** &*f*, int *argc*, char **argv*, std::vector<std::string> **const** &*cfg*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default

partitioner(int *argc*, char **argv*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *runtime_mode* *mode* = runtime_mode_default)

partitioner(int *argc*, char **argv*, std::vector<std::string> *ini_config*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *runtime_mode* *mode* = runtime_mode_default)

partitioner(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char **argv*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *runtime_mode* *mode* = runtime_mode_default)

partitioner(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char **argv*, std::vector<std::string> *ini_config*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *runtime_mode* *mode* = runtime_mode_default)

partitioner(std::nullptr_t *f*, int *argc*, char ***argv*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

partitioner(std::nullptr_t *f*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

partitioner(std::nullptr_t *f*, boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, std::vector<std::string> *ini_config*, *resource*::*partitioner_mode* *rpmode* = resource::mode_default, *runtime_mode* *mode* = runtime_mode_default)

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**

Private Members

detail::partitioner &**partitioner_**

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

persistent_auto_chunk_size()

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*)
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*)
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.

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

const std::size_t **invalid_pu_id** = std::size_t(-1)

Friends

friend hpx::resource::pu::core

friend hpx::resource::pu::numa_domain

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.

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.

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 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 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 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.

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.

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.

struct sequenced_execution_tag
#include <execution_fwd.hpp> Function invocations executed by a group of sequential execution agents execute in sequential order.

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.

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

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<Parameters…>::typ
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.

executor_type **const** &**executor**() **const**
Return the associated executor object.

executor_parameters_type &**parameters**()
Return the associated executor parameters object.

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::sequenced_policy::hpx::serialization::access

template <**typename** Executor, **typename** Parameters>
struct sequenced_policy_shim
#include <execution_policy.hpp> The class *sequenced_policy* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.

Inherits from *hpx::parallel::execution::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

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<Parameters_…>::
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.

Executor **const** &**executor** () **const**
Return the associated executor object.

Parameters &**parameters** ()
Return the associated executor parameters object.

Parameters **const** &**parameters** () **const**
Return the associated executor parameters object.

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

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<Parameters…>::ty
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.

executor_type **const** &**executor**() **const**
Return the associated executor object.

executor_parameters_type &**parameters**()
Return the associated executor parameters object.

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::sequenced_task_policy::hpx::serialization::access

template <**typename** Executor, **typename** Parameters>
struct sequenced_task_policy_shim

#include <execution_policy.hpp> Extension: The class *sequenced_task_policy_shim* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading based on combining a underlying sequenced_task_policy and an executor and indicate that a parallel algorithm's execution may not be parallelized (has to run sequentially).

The algorithm returns a future representing the result of the corresponding algorithm when invoked with the *sequenced_policy*.

Inherits from *hpx::parallel::execution::sequenced_task_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

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<Parameters_…>::
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.

Executor **const** &**executor**() **const**

Return the associated executor object.

Parameters &**parameters**()

Return the associated executor parameters object.

Parameters **const** &**parameters**() **const**

Return the associated executor parameters object.

struct static_chunk_size

#include <static_chunk_size.hpp> Loop iterations are divided into pieces of size *chunk_size* and then assigned to threads. If *chunk_size* is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

Note This executor parameters type is equivalent to OpenMP's STATIC scheduling directive.

Public Functions

static_chunk_size()

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.

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.

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([&] {
            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

#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.

Inherits from *hpx::exception*

Public Functions

task_canceled_exception()

struct thread_interrupted

#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);
            // 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()*.

Inherits from exception

class thread_pool_base

#include <thread_pool_base.hpp> The base class used to manage a pool of OS threads.

Inherits from manage_executor

Public Functions

virtual *hpx*::future<void> **resume**() = 0

Resumes the thread pool. When the all OS threads on the thread pool have been resumed the returned future will be ready.

Note Can only be called from an HPX thread. Use resume_cb or resume_direct to suspend the pool from outside HPX.

Return A `future<void>` which is ready when the thread pool has been resumed.

Exceptions

- `hpx::exception`: if called from outside the HPX runtime.

virtual void **resume_cb**(std::function<void) void

> *callback*, *error_code* *&ec* = throws = 0Resumes the thread pool. Takes a callback as a parameter which will be called when all OS threads on the thread pool have been resumed.

Parameters

- `callback`: [in] called when the thread pool has been resumed.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.

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 *hpx*::future<void> **suspend**() = 0

Suspends the thread pool. When the all OS threads on the thread pool have been suspended the returned future will be ready.

Note Can only be called from an HPX thread. Use suspend_cb or suspend_direct to suspend the pool from outside HPX. A thread pool cannot be suspended from an HPX thread running on the pool itself.

Return A `future<void>` which is ready when the thread pool has been suspended.

Exceptions

- `hpx::exception`: if called from outside the HPX runtime.

virtual void **suspend_cb**(std::function<void) void

> *callback*, *error_code* *&ec* = throws = 0Suspends the thread pool. Takes a callback as a parameter which will be called when all OS threads on the thread pool have been suspended.

Note A thread pool cannot be suspended from an HPX thread running on the pool itself.

Parameters

- `callback`: [in] called when the thread pool has been suspended.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.

Exceptions

- *`hpx::exception`*: if called from an HPX thread which is running on the pool itself.

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.

virtual *hpx*::future<void> **suspend_processing_unit** (std::size_t *virt_core*) = 0
Suspends the given processing unit. When the processing unit has been suspended the returned future will be ready.

Note Can only be called from an HPX thread. Use suspend_processing_unit_cb or to suspend the processing unit from outside HPX. Requires that the pool has *threads::policies::enable_elasticity* set.

Return A `future<void>` which is ready when the given processing unit has been suspended.

Parameters

- `virt_core`: [in] The processing unit on the the pool to be suspended. The processing units are indexed starting from 0.

Exceptions

- *`hpx::exception`*: if called from outside the HPX runtime.

virtual void **suspend_processing_unit_cb** (std::function<void) void
> *callback*, std::size_t *virt_core*, *error_code* &*ec* = throws = 0Suspends the given processing unit. Takes a callback as a parameter which will be called when the processing unit has been suspended.

Note Requires that the pool has *threads::policies::enable_elasticity* set.

Parameters

- `callback`: [in] Callback which is called when the processing unit has been suspended.
- `virt_core`: [in] The processing unit to suspend.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.

virtual *hpx*::future<void> **resume_processing_unit** (std::size_t *virt_core*) = 0
Resumes the given processing unit. When the processing unit has been resumed the returned future will be ready.

Note Can only be called from an HPX thread. Use resume_processing_unit_cb or to resume the processing unit from outside HPX. Requires that the pool has *threads::policies::enable_elasticity* set.

Return A `future<void>` which is ready when the given processing unit has been resumed.

Parameters

- `virt_core`: [in] The processing unit on the the pool to be resumed. The processing units are indexed starting from 0.

virtual void **resume_processing_unit_cb** (std::function<void) void
> *callback*, std::size_t *virt_core*, *error_code* &*ec* = throws = 0Resumes the given processing unit. Takes a callback as a parameter which will be called when the processing unit has been resumed.

Note Requires that the pool has *threads::policies::enable_elasticity* set.

Parameters

- `callback`: [in] Callback which is called when the processing unit has been suspended.
- `virt_core`: [in] The processing unit to resume.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.

struct thread_schedule_hint
#include <thread_enums.hpp>

Public Functions

thread_schedule_hint ()

thread_schedule_hint (std::int16_t *thread_hint*)

thread_schedule_hint (*thread_schedule_hint_mode* *mode*, std::int16_t *hint*)

Public Members

thread_schedule_hint_mode **mode**

std::int16_t **hint**

struct unsequenced_execution_tag
#include <execution_fwd.hpp> Function invocations executed by a group of vector execution agents are permitted to execute in unordered fashion when executed in different threads, and un-sequenced with respect to one another when executed in the same thread.

Note unsequenced_execution_tag is weaker than parallel_execution_tag.

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.

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.

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.

namespace applier
The namespace *applier* contains all definitions needed for the class *hpx::applier::applier* and its related functionality. This namespace is part of the HPX core module.

namespace hpx

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*)

Typedefs

typedef *util*::function_nonser<void (boost::system::error_code **const**&, parcelset::parcel **const**&) > **parcel_write_handler_type**

The type of a function which can be registered as a parcel write handler using the function hpx::set_parcel_write_handler.

Note A parcel write handler is a function which is called by the parcel layer whenever a parcel has been sent by the underlying networking library and if no explicit parcel handler function was specified for the parcel.

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.

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.

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

enum throwmode
Encode error category for new *error_code*.

Values:

plain = 0

rethrow = 1

lightweight = 0x80

enum runtime_mode

A HPX runtime can be executed in two different modes: console mode and worker mode.

Values:

runtime_mode_invalid = -1

runtime_mode_console = 0

The runtime is the console locality.

runtime_mode_worker = 1

The runtime is a worker locality.

runtime_mode_connect = 2

The runtime is a worker locality connecting late

runtime_mode_default = 3

The runtime mode will be determined based on the command line arguments

runtime_mode_last

Functions

int **init** (*util*::function_nonser<int) boost::program_options::variables_map &vm
> **const** &*f*, boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

Return The function returns the value, which has been returned from the user supplied `f`.

Note If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.

- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init** (int (**f*)) boost::program_options::variables_map &vm , boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

Return The function returns the value, which has been returned from the user supplied `f`.

Note If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.
- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init** (boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)
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).

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.
- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

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).

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.
- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

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).

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

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`.

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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode*)

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).

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

Note If the parameter `mode` is runtime_mode_default, 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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode* *mode*)

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).

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

Note If the parameter `mode` is runtime_mode_default, 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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')

- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(std::string **const** &*app_name*, int *argc* = 0, char ***argv* = nullptr, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

Main entry point for launching the HPX runtime system.

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`.

Parameters

- `app_name`: [in] The name of the application.
- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(int *argc* = 0, char ***argv* = nullptr, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

Main entry point for launching the HPX runtime system.

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. If not command line arguments are passed, console mode is assumed.

Note If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section 'HPX Command Line Options'.

Parameters

- `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()`).

- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

Main entry point for launching the HPX runtime system.

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. If not command line arguments are passed, console mode is assumed.

Note If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section 'HPX Command Line Options'.

Parameters

- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init**(int (**f*)) boost::program_options::variables_map &vm

, std::string **const** &*app_name*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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). It will schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

Return The function returns the value, which has been returned from the user supplied function `f`.

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`.

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`.
- `app_name`: [in] The name of the application.
- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init** (int (*f*)) boost::program_options::variables_map &vm
, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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). It will schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

Return The function returns the value, which has been returned from the user supplied function `f`.

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`.

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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init** (*util*::function_nonser<int) int, char **
> **const** &*f*, std::string **const** &*app_name*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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). It will schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

Return The function returns the value, which has been returned from the user supplied function `f`.

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`.

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`.
- `app_name`: [in] The name of the application.

- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init** (*util*::function_nonser<int) int, char **
> **const** &*f*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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). It will schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

Return The function returns the value, which has been returned from the user supplied function `f`.

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`.

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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

int **init** (*util*::function_nonser<int) int, char **
> **const** &*f*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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). It will schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

Return The function returns the value, which has been returned from the user supplied function `f`.

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`.

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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(*util*::function_nonser<int) boost::program_options::variables_map &vm > **const** &*f*, boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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`.

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`.
- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')

- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.
- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start** (int (*f*)) boost::program_options::variables_map &vm

, boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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`.

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`.
- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.
- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

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.

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.
- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *startup_function_type* *startup* = startup_function_type (), *shutdown_function_type* *shutdown* = shutdown_function_type (), *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

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.

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `startup`: [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed.
- `shutdown`: [in] A function to be executed inside an HPX thread while *hpx::finalize* is executed. If this parameter is not given no function will be executed.
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

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.

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

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 runtime_mode_default, 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()`).

- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode*)

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.

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

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 runtime_mode_default, 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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(boost::program_options::options_description **const** &*desc_cmdline*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode* *mode*)

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.

In console mode it will execute the user supplied function `hpx_main`, in worker mode it will execute an empty `hpx_main`.

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 runtime_mode_default, 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

- `desc_cmdline`: [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by hpx::init (see description below).
- `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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(std::string **const** &*app_name*, int *argc* = 0, char ***argv* = nullptr, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)

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.

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`.

Parameters

- `app_name`: [in] The name of the application.
- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(int *argc* = 0, char ***argv* = nullptr, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)
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.

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

- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start**(std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_default)
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.

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

- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode

(hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start** (int (*f)) boost::program_options::variables_map &vm
, std::string **const** &*app_name*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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 overload will schedule the function given by `f` as a HPX thread. It will not call `hpx_main`.

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`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application.
- `app_name`: [in] The name of the application.
- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start** (*util*::function_nonser<int) int, char **
> **const** &*f*, std::string **const** &*app_name*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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 overload will schedule the function given by `f` as a HPX thread. It will not call `hpx_main`.

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`.

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`.

- `app_name`: [in] The name of the application.
- `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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start** (int (**f*)) boost::program_options::variables_map &vm
, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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 overload will schedule the function given by `f` as a HPX thread. It will not call `hpx_main`.

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`.

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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start** (*util*::function_nonser<int) int, char **
> **const** &*f*, int *argc*, char ***argv*, *hpx*::*runtime_mode* *mode* = hpx::runtime_mode_defaultMain 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 overload will schedule the function given by `f` as a HPX thread. It will not call `hpx_main`.

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`.

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()`).
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

bool **start** (*util*::function_nonser<int) int, char **
> **const** &*f*, int *argc*, char ***argv*, std::vector<std::string> **const** &*cfg*, *hpx*::*runtime_mode mode* = hpx::runtime_mode_defaultMain 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 overload will schedule the function given by `f` as a HPX thread. It will not call `hpx_main`.

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`.

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()`).
- `cfg`: A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1')
- `mode`: [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (hpx::runtime_mode_console), all other localities have to be run in worker mode (hpx::runtime_mode_worker). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode.

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`).

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.

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).

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::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)

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`:

error **get_error**(*hpx*::*exception* **const** &*e*)
Return the locality id where the exception was 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 locality id where the exception was 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_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_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_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::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`:

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)

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 silently fail.

void **unregister_thread**(runtime **rt*)
Unregister the thread from HPX, this should be done once in the end before the external thread exists.

naming::gid_type **const** &**get_locality**()
The function *get_locality* returns a reference to the locality prefix.

std::size_t **get_runtime_instance_number**()
The function *get_runtime_instance_number* returns a unique 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.

void **start_active_counters**(*error_code* &*ec* = throws)
Start all active performance counters, optionally naming the section of code.

Note As long as *ec* is not pre-initialized to hpx::throws this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note The active counters are those which have been specified on the command line while executing the application (see command line option –hpx:print-counter)

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 **reset_active_counters**(*error_code* &*ec* = throws)
Resets all active performance counters.

Note As long as *ec* is not pre-initialized to hpx::throws this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note The active counters are those which have been specified on the command line while executing the application (see command line option –hpx:print-counter)

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 **reinit_active_counters**(bool *reset* = true, *error_code* &*ec* = throws)
Re-initialize all active performance counters.

Note As long as *ec* is not pre-initialized to hpx::throws this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note The active counters are those which have been specified on the command line while executing the application (see command line option –hpx:print-counter)

Parameters

- `reset`: [in] Reset the current values before re-initializing counters (default: true)
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.

void **stop_active_counters**(*error_code* &*ec* = throws)

Stop all active performance counters.

Note As long as *ec* is not pre-initialized to hpx::throws this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note The active counters are those which have been specified on the command line while executing the application (see command line option –hpx:print-counter)

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 **evaluate_active_counters**(bool *reset* = false, char **const** **description* = nullptr, *error_code* &*ec* = throws)

Evaluate and output all active performance counters, optionally naming the point in code marked by this function.

Note As long as *ec* is not pre-initialized to hpx::throws this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note The output generated by this function is redirected to the destination specified by the corresponding command line options (see –hpx:print-counter-destination).

Note The active counters are those which have been specified on the command line while executing the application (see command line option –hpx:print-counter)

Parameters

- `reset`: [in] this is an optional flag allowing to reset the counter value after it has been evaluated.
- `description`: [in] this is an optional value naming the point in the code marked by the call to this function.
- `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.

serialization::binary_filter ***create_binary_filter**(char **const** **binary_filter_type*, bool *compress*, serialization::binary_filter **next_filter* = nullptr, *error_code* &*ec* = throws)

Create an instance of a binary filter plugin.

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

- `binary_filter_type`: [in] The type of the binary filter to create
- `compress`: [in] The created filter should support compression
- `next_filter`: [in] Use this as the filter to dispatch the invocation into.
- `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 **`is_networking_enabled`**()

Return true if networking is enabled.

Note Networking is enabled if `-DHPX_WITH_NETWORKING=On` was used at configuration time and more than one locality is used or the command line option `--hpx:expect-connecting-localities` was specified

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* = ~0U)

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* = ~0U)

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* = ~0U)

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* = ~0U)

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* = ~0U)

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::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.

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

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.

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.

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>".

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.

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.

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*)
Get the internal representation (runtime_mode constant) from the readable string representing the name.

parcel_write_handler_type **set_parcel_write_handler**(*parcel_write_handler_type* **const** &*f*)
Set the default parcel write handler which is invoked once a parcel has been sent if no explicit write handler was specified.

Return The function returns the parcel write handler which was installed before this function was called.

Note If no parcel handler function is registered by the user the system will call a default parcel handler function which is not performing any actions. However, this default function will terminate the application in case of any errors detected during preparing or sending the parcel.

Parameters

- `f`: The new parcel write handler to use from this point on

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.

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()

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*::i **const** &*id*, Re-sult &&*t*, bool *move_cre* = 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_unmana**

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*::i **const** &*id*, Result &&*t*, *naming*::id_ty **const** &*cont*, bool *move_cre* = 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_unmana**

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 & locality, Ts &&... vs)

Create one or more new instances of the given Component type on the specified locality.

This function creates one or more new instances of the given Component type on the specified locality and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
   hpx::new_<some_component>(hpx::find_here(), ...);
hpx::id_type id = f.get();
```

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 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 funtion 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 & locality, std::size_t count, Ts &&... vs)

Create multiple new instances of the given Component type on the specified locality.

This function creates multiple new instances of the given Component type on the specified locality and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type> > f =
   hpx::new_<some_component[]>(hpx::find_here(), 10, ...);
hpx::id_type id = f.get();
```

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::new_(DistPolicy const & policy, Ts &&... vs)

Create one or more new instances of the given Component type based on the given distribution policy.

This function creates one or more new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for global address which can be used to reference the new component instance(s).

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
   hpx::new_<some_component>(hpx::default_layout, ...);
hpx::id_type id = f.get();
```

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::new_(DistPolicy const & policy, std::size_t count, Ts &&... vs)

Create multiple new instances of the given Component type on the localities as defined by the given distribution policy.

This function creates multiple new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type> > f =
   hpx::new_<some_component[]>(hpx::default_layout, 10, ...);
hpx::id_type id = f.get();
```

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.

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

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*.

template <typename InputIter, **typename** Container = vector<future<typename std::iterator_traits<InputIter>::value_type
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_type
future<Container> **when_all_n**(InputIter *begin*, std::size_t *count*)

The function *when_all_n* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after they finished executing.

Return Returns a future holding the same list of futures as has been passed to *when_all_n*.

- future<Container<future<R>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output vector will be the same as given by the input iterator.

Note As long as *ec* is not pre-initialized to hpx::throws this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note None of the futures in the input sequence are invalidated.

Parameters

- `begin`: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_all_n* should wait.
- `count`: [in] The number of elements in the sequence starting at *first*.

Exceptions

- `This`: function will throw errors which are encountered while setting up the requested operation only. Errors encountered while executing the operations delivering the results to be stored in the futures are reported through the futures themselves.

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 **wait_any**(std::array<future<R>, N> &*futures*, *error_code* &*ec* = throws)

The function *wait_any* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function *wait_any* returns after at least one future has become ready. All input futures are still valid after *wait_any* returns.

Note As long as *ec* is not pre-initialized to hpx::throws this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note None of the futures in the input sequence are invalidated.

Parameters

- `futures`: [in] 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.

template <typename InputIter, **typename** Container = vector<future<typename std::iterator_traits<InputIter>::value_type
future<*when_any_result*<Container>> **when_any**(InputIter *first*, InputIter *last*)

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_type
future<*when_any_result*<Container>> **when_any_n**(InputIter *first*, std::size_t *count*)

The function *when_any_n* is a non-deterministic choice operator. It OR-composes all future objects given and returns a new future object representing the same list of futures after one future of that list finishes execution.

Return Returns a *when_any_result* holding the same list of futures as has been passed to when_any and an index pointing to a ready future.

- future<*when_any_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

Note None of the futures in the input sequence are invalidated.

Parameters

- `first`: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_any_n* should wait.
- `count`: [in] The number of elements in the sequence starting at *first*.

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.

template <typename InputIter, **typename** Container = vector<future<typename std::iterator_traits<InputIter>::value_type
future<*when_some_result*<Container>> **when_some** (std::size_t *n*, Iterator *first*, Iterator *last*, *error_code* &*ec* = throws)

The function *when_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The future returned by the function *when_some* becomes ready when at least *n* argument futures have become ready.

Return Returns a *when_some_result* holding the same list of futures as has been passed to when_some and indices pointing to ready futures.

- future<*when_some_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

Note Calling this version of *when_some* where first == last, returns a future with an empty container that is immediately ready. Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some* will not throw an exception, but the futures held in the output collection may.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `first`: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- `last`: [in] The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.

template <typename Range>
future<*when_some_result*<Range>> **when_some**(std::size_t *n*, Range &&*futures*, *error_code* &*ec* = throws)

The function *when_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The future returned by the function *when_some* becomes ready when at least *n* argument futures have become ready.

Return Returns a *when_some_result* holding the same list of futures as has been passed to when_some and indices pointing to ready futures.

- future<*when_some_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

Note Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some* will not throw an exception, but the futures held in the output collection may.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `futures`: [in] A container holding an arbitrary amount of *future* or *shared_future* objects for which *when_some* should wait.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.

template <typename ... *T*>
future<*when_some_result*<tuple<future<T>...>>> **when_some**(std::size_t *n*, *error_code* &*ec*, T&&... *futures*)

The function *when_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The future returned by the function *when_some* becomes ready when at least *n* argument futures have become ready.

Return Returns a *when_some_result* holding the same list of futures as has been passed to when_some and an index pointing to a ready future..

- future<*when_some_result*<tuple<future<T0>, future<T1>…>>>: If inputs are fixed in number and are of heterogeneous types. The inputs can be any arbitrary number of future objects.

- future<*when_some_result*<tuple<>>> if *when_some* is called with zero arguments. The returned future will be initially ready.

Note Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some* will not throw an exception, but the futures held in the output collection may.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::throws the function will throw on error instead.
- `futures`: [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *when_some* should wait.

template <**typename**... *T*>
future<*when_some_result*<tuple<future<T>...>>> **when_some**(std::size_t *n*, T&&... *futures*)

The function *when_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The future returned by the function *when_some* becomes ready when at least *n* argument futures have become ready.

Return Returns a *when_some_result* holding the same list of futures as has been passed to when_some and an index pointing to a ready future..

- future<*when_some_result*<tuple<future<T0>, future<T1>…>>>: If inputs are fixed in number and are of heterogeneous types. The inputs can be any arbitrary number of future objects.
- future<*when_some_result*<tuple<>>> if *when_some* is called with zero arguments. The returned future will be initially ready.

Note Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some* will not throw an exception, but the futures held in the output collection may.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `futures`: [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *when_some* should wait.

template <**typename** InputIter, **typename** Container = vector<future<typename std::iterator_traits<InputIter>::value_type
future<*when_some_result*<Container>> **when_some_n**(std::size_t *n*, Iterator *first*, std::size_t *count*, *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** 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*.

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*.

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.

namespace actions

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 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 lcos

Functions

template <**typename** Action, **typename** ArgN, . . . >
hpx::future<std::vector<decltype(Action(hpx::id_type, ArgN, ...))>> **broadcast** (std::vector<*hpx*::id_type> **const** &*ids*, ArgN *argN*, ...)

Perform a distributed broadcast operation.

The function *hpx::lcos::broadcast* performs a distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

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 **broadcast_apply**(std::vector<*hpx*::id_type> **const** &*ids*, ArgN *argN*, ...)

Perform an asynchronous (fire&forget) distributed broadcast operation.

The function *hpx::lcos::broadcast_apply* performs an asynchronous (fire&forget) distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `argN`: [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

template <**typename** Action, **typename** ArgN, …>
hpx::future<std::vector<decltype(Action(hpx::id_type, ArgN, ..., std::size_t))>> **broadcast_with_index**(std::vector<*hpx*::i **const** &*ids*, ArgN *argN*, ...)

Perform a distributed broadcast operation.

The function *hpx::lcos::broadcast_with_index* performs a distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Return This function returns a future representing the result of the overall reduction operation.

Note If decltype(Action(…)) is void, then the result of this function is future<void>.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `argN`: [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

template <**typename** Action, **typename** ArgN, …>

void **broadcast_apply_with_index**(std::vector<*hpx*::id_type> **const** &*ids*, ArgN *argN*, ...)

Perform an asynchronous (fire&forget) distributed broadcast operation.

The function *hpx::lcos::broadcast_apply_with_index* performs an asynchronous (fire&forget) distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `argN`: [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

template <**typename** Action, **typename** FoldOp, **typename** Init, **typename** ArgN, ...>
hpx::future<decltype(Action(hpx::id_type, ArgN, ...))> **fold**(std::vector<*hpx*::id_type> **const** &*ids*, FoldOp &&*fold_op*, Init &&*init*, ArgN *argN*, ...)

Perform a distributed fold operation.

The function *hpx::lcos::fold* performs a distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

Note The type of the initial value must be convertible to the result type returned from the invoked action.

Return This function returns a future representing the result of the overall folding operation.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `fold_op`: [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- `init`: [in] The initial value to be used for the folding operation
- `argN`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

template <**typename** Action, **typename** FoldOp, **typename** Init, **typename** ArgN, ...>
hpx::future<decltype(Action(hpx::id_type, ArgN, ..., std::size_t))> **fold_with_index**(std::vector<*hpx*::id_type> **const** &*ids*, FoldOp &&*fold_op*, Init &&*init*, ArgN *argN*, ...)

Perform a distributed folding operation.

The function *hpx::lcos::fold_with_index* performs a distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note The type of the initial value must be convertible to the result type returned from the invoked action.

Return This function returns a future representing the result of the overall folding operation.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `fold_op`: [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- `init`: [in] The initial value to be used for the folding operation
- `argN`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

template <**typename** Action, **typename** FoldOp, **typename** Init, **typename** ArgN, ...>
hpx::future<decltype(Action(hpx::id_type, ArgN, ...))> **inverse_fold**(std::vector<*hpx*::id_type> **const** &*ids*, FoldOp &&*fold_op*, Init &&*init*, ArgN *argN*, ...)

Perform a distributed inverse folding operation.

The function *hpx::lcos::inverse_fold* performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

Note The type of the initial value must be convertible to the result type returned from the invoked action.

Return This function returns a future representing the result of the overall folding operation.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `fold_op`: [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- `init`: [in] The initial value to be used for the folding operation
- `argN`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

template <**typename** Action, **typename** FoldOp, **typename** Init, **typename** ArgN, ...>

hpx::future<decltype(Action(hpx::id_type, ArgN, ..., std::size_t))> **inverse_fold_with_index**(std::vector<*hpx*::id_type> **const** &*ids*, FoldOp &&*fold_op*, Init &&*init*, ArgN *argN*, ...)

Perform a distributed inverse folding operation.

The function *hpx::lcos::inverse_fold_with_index* performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note The type of the initial value must be convertible to the result type returned from the invoked action.

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** 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<void> **gather_there**(char **const** **basename*, *hpx*::future<T> *result*, std::size_t *generation* = std::size_t(-1), std::size_t *root_site* = 0, std::size_t *this_site* = std::size_t(-1))

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 which 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.
- `root_site`: The sequence number of the central gather point (usually the locality id). This value is optional and defaults to 0.
- `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_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<void> **gather_there**(char **const** **basename*, T &&*result*, std::size_t *generation* = std::size_t(-1), std::size_t *root_site* = 0, std::size_t *this_site* = std::size_t(-1))

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 which 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.
- `root_site`: The sequence number of the central gather point (usually the locality id). This value is optional and defaults to 0.
- `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** Action, **typename** ReduceOp, **typename** ArgN, …>
hpx::future<decltype(Action(hpx::id_type, ArgN, ...))> **reduce**(std::vector<*hpx*::id_type> **const** &*ids*, ReduceOp &&*reduce_op*, ArgN *argN*, ...)

Perform a distributed reduction operation.

The function *hpx::lcos::reduce* performs a distributed reduction operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

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<decltype(Action(hpx::id_type, ArgN, ..., std::size_t))> **reduce_with_index**(std::vector<*hpx*::id_type> **const** &*ids*, ReduceOp &&*reduce_op*, ArgN *argN*, ...)

Perform a distributed reduction operation.

The function *hpx::lcos::reduce_with_index* performs a distributed reduction operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

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.

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.

namespace parallel

namespace execution

Typedefs

using hpx::parallel::execution::parallel_executor = typedef parallel_policy_executor<hp

using hpx::parallel::execution::service_executor = typedef threads::executors::service_

A *service_executor* exposes one of the predefined HPX thread pools through an executor interface.

Note All tasks executed by one of these executors will run on one of the OS-threads dedicated for the given thread pool. The tasks will not run as HPX-threads.

using hpx::parallel::execution::io_pool_executor = typedef threads::executors::io_pool_

A *io_pool_executor* exposes the predefined HPX IO thread pool through an executor interface.

Note All tasks executed by one of these executors will run on one of the OS-threads dedicated for the IO thread pool. The tasks will not run as HPX-threads.

using hpx::parallel::execution::parcel_pool_executor = typedef threads::executors::parc

A *io_pool_executor* exposes the predefined HPX parcel thread pool through an executor interface.

Note All tasks executed by one of these executors will run on one of the OS-threads dedicated for the parcel thread pool. The tasks will not run as HPX-threads.

using hpx::parallel::execution::timer_pool_executor = typedef threads::executors::time

A *io_pool_executor* exposes the predefined HPX timer thread pool through an executor interface.

Note All tasks executed by one of these executors will run on one of the OS-threads dedicated for the timer thread pool. The tasks will not run as HPX-threads.

using hpx::parallel::execution::main_pool_executor = typedef threads::executors::main_

A *io_pool_executor* exposes the predefined HPX main thread pool through an executor interface.

Note All tasks executed by one of these executors will run on one of the OS-threads dedicated for the main thread pool. The tasks will not run as HPX-threads.

using hpx::parallel::execution::local_priority_queue_executor = typedef threads::execu

Creates a new local_priority_queue_executor

Parameters

- `max_punits`: [in] The maximum number of processing units to associate with the newly created executor.
- `min_punits`: [in] The minimum number of processing units to associate with the newly created executor (default: 1).

Variables

task_policy_tag HPX_CONSTEXPR_OR_CONST hpx::parallel::execution::task

Default sequential execution policy object.

HPX_STATIC_CONSTEXPR sequenced_policy hpx::parallel::execution::seq

Default sequential execution policy object.

HPX_STATIC_CONSTEXPR parallel_policy hpx::parallel::execution::par

Default parallel execution policy object.

HPX_STATIC_CONSTEXPR parallel_unsequenced_policy hpx::parallel::execution::par_unseq

Default vector execution policy object.

namespace

namespace

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, FwdIter2>:

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, FwdIter2>:

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.

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, FwdIter>::t

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.

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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2>
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (FwdIter1), tag::out 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<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** Size, **typename** FwdIter2>
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (FwdIter1) , tag::out 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<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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** F, **typename** Proj = util::projec
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (FwdIter1), tag::out
FwdIter2>>::type **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 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.
- `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 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.
- `proj`: Specifies 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<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.

template <**typename** ExPolicy, **typename** FwdIter, **typename** T, **typename** Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, **typename** std::iterator_traits<FwdIter>::difference_type>::type **count**(ExPolicy &&*policy*, FwdIter *first*, FwdIter *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.
- `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 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<FwdIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template <**typename** ExPolicy, **typename** FwdIter, **typename** F, **typename** Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, **typename** std::iterator_traits<FwdIter>::difference_type>::type **count_if** (ExPolicy &&*policy*, FwdIter *first*, FwdIter *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<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.
- `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 *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 *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.

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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** Pred = detail::equal_to>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, bool>::type

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>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, bool>::type

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.

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, FwdIter2>:

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 *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 *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, FwdIter2>:

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 *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 *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.

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*.

template <**typename** ExPolicy, **typename** FwdIter, **typename** T>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter>::t

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>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter>::t

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>
std::enable_if<execution::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter>::t
```

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, typena
```

std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter1>:

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, **typena**
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter1>:

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*.

template <typename ExPolicy, **typename** FwdIter, **typename** Size, **typename** F, **typename** Proj = util::projection_i
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.

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*.
- `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.
- `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.
- `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 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.

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

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.
- `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.
- `first`: Refers to the beginning of the sequence 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.

- `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 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** 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*.

template <**typename** ExPolicy, **typename** RandIter, **typename** Comp = detail::less, **typename** Proj = util::projection_
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::projection_
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.

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, bool>::type

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.

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 *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 *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 *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 *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, FwdIter2>:

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 *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 *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.

template <**typename** ExPolicy, **typename** FwdIter, **typename** Pred>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, bool>::type

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.

template <**typename** ExPolicy, **typename** FwdIter, **typename** Pred = detail::less>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, bool>::type

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, FwdIter>::t

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.

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, bool>::type

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.

template <typename ExPolicy, **typename** RandIter1, **typename** RandIter2, **typename** RandIter3, **typename** Comp =
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_tuple<tag::in1 (RandIter1) , tag::in2
RandIter2, tag::outRandIter3>>::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::projection_
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*

template <**typename** ExPolicy, **typename** FwdIter, **typename** Proj = util::projection_identity, **typename** F = detail::le
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 = detail::le
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 = detail::le
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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** Pred = detail::equal_to>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, std::pair<F

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>

std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, std::pair<F

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).

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2>
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (FwdIter1) , tag::out 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.

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** Pred, **typ**
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_tuple<tag::in(FwdIter1), tag::out1FwdIter2, tag::out2FwdIter3>>::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.

template <**typename** ExPolicy, **typename** FwdIter, **typename** T, **typename** F>

std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, T>::type>::

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 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>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, T>::type>::

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 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>
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, **typename**

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 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.

template <**typename** ExPolicy, **typename** RanIter, **typename** RanIter2, **typename** FwdIter1, **typename** FwdIter2, **t**

util::detail::algorithm_result<ExPolicy, std::pair<FwdIter1, FwdIter2>>::type **reduce_by_key**(ExPolicy &&*policy*, RanIter *key_first*, RanIter *key_last*, RanIter2 *values_first*, FwdIter1 *keys_output*, FwdIter2 *values_output*, Compare &&*comp* = Compare(), Func &&*func* = Func())

Reduce by Key performs an inclusive scan reduction operation on elements supplied in key/value pairs. The algorithm produces a single output value for each set of equal consecutive keys in [key_first, key_last). the value being the GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))). for the run of consecutive matching keys. The number of keys supplied must match the number of values.

comp has to induce a strict weak ordering on the values.

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

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `RanIter`: The type of the key iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- `RanIter2`: The type of the value iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- `FwdIter1`: The type of the iterator representing the destination key range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination value range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Compare`: The type of the optional function/function object to use to compare keys (deduced). Assumed to be std::equal_to otherwise.

- `Func`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `key_first`: Refers to the beginning of the sequence of key elements the algorithm will be applied to.
- `key_last`: Refers to the end of the sequence of key elements the algorithm will be applied to.
- `values_first`: Refers to the beginning of the sequence of value elements the algorithm will be applied to.
- `keys_output`: Refers to the start output location for the keys produced by the algorithm.
- `values_output`: Refers to the start output location for the values produced by the algorithm.
- `comp`: comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- `func`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  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.

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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** T, **typename** Proj = util::projec
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (FwdIter1), tag::out
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 *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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** F, **typename** Proj = util::projec
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (FwdIter1) , tag::out
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.

template <**typename** ExPolicy, **typename** FwdIter, **typename** T1, **typename** T2, **typename** Proj = util::projection_i
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_iden
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** Pr
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.

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, *hpx*::*util*::tagged_pair<tag::in(BidirIter), tag::out
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.

template <**typename** ExPolicy, **typename** FwdIter>
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::begin(FwdIter), tag::end 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, *hpx*::*util*::tagged_pair<tag::in (FwdIter1) , tag::out 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.

template <**typename** ExPolicy, **typename** FwdIter, **typename** FwdIter2, **typename** Pred = detail::equal_to, **typenam**
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, **typenam**
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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** FwdIter3, **typename** Pred = de
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter3>:

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.

template <typename ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** FwdIter3, **typename** Pred = de

std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter3>:

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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** FwdIter3, **typename** Pred = de
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter3>:

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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** FwdIter3, **typename** Pred = de
std::enable_if<*execution*::*is_execution_policy*<ExPolicy>::value, **typename** *util*::detail::algorithm_result<ExPolicy, FwdIter3>:

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.

template <typename ExPolicy, **typename** RandomIt, **typename** Proj = util::projection_identity, **typename** Compare =
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.

`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.

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

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*.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** F, **typename** Proj = util::projec
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, **typen**
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_tuple<tag::in1 (FwdIter1), tag::in2
FwdIter2, tag::outFwdIter3>>::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, **typen**
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_tuple<tag::in1 (FwdIter1) , tag::in2
FwdIter2, tag::outFwdIter3>>::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.

template <typename ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** T, **typename** Op, **typename** C
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 *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 *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.

template <**typename** ExPolicy, **typename** FwdIter1, **typename** FwdIter2, **typename** Op, **typename** Conv, **typenar**
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 *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 *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 *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 *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.

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, **typena**
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.

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

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, FwdIter2>:

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. 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.

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.

```
template <typename ExPolicy, typename FwdIter, typename T>
std::enable_if<execution::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy>::type>::typ
```

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>::type>::ty

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.

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.

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.

template <**typename** ExPolicy, **typename** FwdIter, **typename** Pred = detail::equal_to, **typename** Proj = util::projectio
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, **typena**
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.

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.

template <**typename** ExPolicy, **typename** Rng, **typename** OutIter>
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (**typename** *hpx*::*traits*::range_traits<Rng>::iterator_type), tag::out OutIter>>::type **copy**ExPolicy &&*policy*, Rng &&*rng*, OutIter *dest*Copies the elements in the range *rng* to another range beginning at *dest*.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `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.
- `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<tagged_pair<tag::in(iterator_t<Rng>)*, tag::out(FwdIter2)> > if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(iterator_t<Rng>)*, 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** Rng, **typename** OutIter, **typename** F, **typename** Proj = util::projection_id
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (**typename** *hpx*::*traits*::range_traits<Rng>::iterator_type), tag::out OutIter>>::type **copy_if**ExPolicy &&*policy*, Rng &&*rng*, OutIter *dest*, F &&*f*, Proj &&*proj* = Proj()Copies the elements in the range *rng* to another range beginning at *dest*. Copies only the elements for which the predicate *f* returns true. The order of the elements that are not removed is preserved.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than std::distance(begin(rng), end(rng)) assignments, exactly std::distance(begin(rng), end(rng)) applications of the predicate *f*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- `Proj`: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *copy_if* algorithm returns a *hpx::future<tagged_pair<tag::in(iterator_t<Rng>)*, tag::out(FwdIter2)> > if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(iterator_t<Rng>)*, tag::out(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** T, **typename** Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, **typename** std::iterator_traits<**typename** *hpx*::*traits*::range_traits<Rng>::iterator_type

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_type

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.

template <**typename** ExPolicy, **typename** Rng, **typename** T>
util::detail::algorithm_result<ExPolicy>::type **fill**(ExPolicy &&*policy*, Rng &&*rng*, 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.
- `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** Rng, **typename** Size, **typename** T>
util::detail::algorithm_result<ExPolicy, **typename** *hpx*::*traits*::range_traits<Rng>::iterator_type>::type **fill_n**(ExPolicy &&*policy*, Rng &*rng*, 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.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input 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.
- `rng`: Refers to 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*.

`template <typename ExPolicy, typename Rng, typename Rng2, typename Pred = detail::equal_to, typename Proj` *util*::detail::algorithm_result<ExPolicy, **typename** *hpx*::*traits*::range_iterator<Rng>::type>::type **find_end** (ExPolicy &&*policy*, Rng &&*rng*, Rng2 &&*rng2*, Pred &&*op* = Pred(), Proj &&*proj* = Proj())

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.
- `Rng`: 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<>
- `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 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.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng>* and 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 = detail::equal_to, **typename** Pro
util::detail::algorithm_result<ExPolicy, **typename** *hpx*::*traits*::range_iterator<Rng1>::type>::type **find_first_of**(ExPolicy &&*policy*, Rng1 &&*rng1*, Rng2 &&*rng2*, Pred &&*op* = Pred(), Proj1 &&*proj1* = Proj1(), Proj2 &&*proj2* = Proj2())

Searches the range *rng1* for any elements in the range *rng2*. Uses binary predicate *p* to compare elements

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most (S*N) comparisons where *S* = 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.

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<InIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *InIter* otherwise. It returns *last*.

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** Rng, **typename** Comp = detail::less, **typename** Proj = util::projection_ident
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_ident
util::detail::algorithm_result<ExPolicy, **typename** *hpx*::*traits*::range_iterator<Rng>::type>::type **is_heap_until**(ExPolicy &&*policy*, Rng &&*rng*, Comp &&*comp* = Comp(), Proj &&*proj* = Proj())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of 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.

template <**typename** ExPolicy, **typename** Rng1, **typename** Rng2, **typename** RandIter3, **typename** Comp = detail::l
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::outRandIter3>>::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** Proj
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*

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_element** (ExPoli &&*policy*, Rng &&*rng*, F &&*f* = F(), Proj &&*proj* = Proj())

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *min_element* requires *F* to meet the requirements of *CopyConstructible*.
- `Proj`: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

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

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *min_element* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template <**typename** ExPolicy, **typename** Rng, **typename** Proj = util::projection_identity, **typename** F = detail::less>
util::detail::algorithm_result<ExPolicy, **typename** *hpx*::*traits*::range_traits<Rng>::iterator_type>::type **max_element** (ExPoli &&*po icy*, Rng &&*rn* F &&*f* = F(), Proj &&*pr* = Proj())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *max_element* requires *F* to meet the requirements of *CopyConstructible*.
- `Proj`: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the 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.

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, **typena**
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_tuple<tag::in(**typename**
hpx::*traits*::range_iterator<Rng>::type),
tag::out1
FwdIter2, tag::out2FwdIter3>>::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.

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**(ExPolicy &&*policy*, Rng &&*rng*, 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.
- `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.

template <**typename** ExPolicy, **typename** Rng, **typename** OutIter, **typename** T, **typename** Proj = util::projection_id
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (**typename**
hpx::*traits*::range_traits<Rng>::iterator_type),
tag::out
OutIter>>::type **remove_copy**ExPolicy &&*policy*, Rng &&*rng*, OutIter *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.
- `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_id
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::in (**typename**
hpx::*traits*::range_traits<Rng>::iterator_type),
tag::out
OutIter>>::type **remove_copy_if**ExPolicy &&*policy*, Rng &&*rng*, OutIter *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.
- `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.

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**(ExPolicy &&*policy*, Rng &&*rng*, T1 **const** &*old_value*, T2 **const** &*new_value*, Proj &&*proj* = Proj())

Replaces all elements satisfying specific criteria with *new_value* in the range [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**(ExPolicy &&*policy*, Rng &&*rng*, 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

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 = uti
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.

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, *hpx*::*util*::tagged_pair<tag::in(**typename** *hpx*::*traits*::range_iterator<Rng>::type), tag::out
OutIter>>::type **reverse_copy**ExPolicy &&*policy*, Rng &&*rng*, OutIter *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.
- `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<tagged_pair<tag::in(BidirIter), tag::out(OutIter)> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(BidirIter), 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>
util::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_pair<tag::begin(**typename** *hpx*::*traits*::range_iterator<Rng>::type), tag::end
typename *hpx*::*traits*::range_iterator<Rng>::type>>::type **rotate**ExPolicy &&*policy*, Rng &&*rng*, **typename** *hpx*::*traits*::range_iterator<Rng>::type *middle*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, *hpx*::*util*::tagged_pair<tag::in (**typename** *hpx*::*traits*::range_iterator<Rng>::type), tag::out OutIter>>::type **rotate_copy**ExPolicy &&*policy*, Rng &&*rng*, **typename** *hpx*::*traits*::range_iterator<Rng>::type *middle*, OutIter *dest_first*Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *new_first* becomes the first element of the new range and *new_first* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm 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.

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**(ExPolicy &&*policy*, Rng1 &&*rng1*, std::size_t *count*, Rng2 &&*rng2*, Pred &&*op* = Pred(), Proj1 &&*proj1* = Proj1(), Proj2 &&*proj2* = Proj2())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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.

template <typename ExPolicy, **typename** Rng, **typename** Proj = util::projection_identity, **typename** Compare = detai

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*.

template <**typename** ExPolicy, **typename** Rng, **typename** OutIter, **typename** F, **typename** Proj = util::projection_id
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** Proj *util*::detail::algorithm_result<ExPolicy, *hpx*::*util*::tagged_tuple<tag::in1 (**typename** *hpx*::*traits*::range_iterator<Rng>::type), tag::in2 InIter2, tag::outOutIter>>::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** Proj
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::outOutIter>>::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.

template <**typename** ExPolicy, **typename** Rng, **typename** Pred = detail::equal_to, **typename** Proj = util::projection_ic
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** P
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.

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, **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… *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`** S, typename… *Args*, &&std::is_integral< S >::value>
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** S, typename… *Args*, &&std::is_integral< S >::value>
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** Size, typename… *Args*, &&std::is_integral< Size >::value>
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… *Args*, &&std::is_integral< Size >::value>
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** S, typename... *Args*, &&std::is_integral< Siz *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.

template <**typename** I, **typename** Size, **typename** S, typename… *Args*, &&std::is_integral< Size >::value &&std::is_i
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** T>
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)

template <**typename** T, **typename** Op>
detail::reduction_helper<T, **typename** std::decay<Op>::type> **reduction**(T &*var*, T **const** &*identity*, Op &&*combiner*)

The function template returns a reduction object of unspecified type having a value type and encapsulating an identity value for the reduction, a combiner function object, and a live-out object from which the initial value is obtained and into which the final value is stored.

A parallel algorithm uses reduction objects by allocating an unspecified number of instances, called views, of the reduction's value type. Each view is initialized with the reduction object's identity value, except that the live-out object (which was initialized by the caller) comprises one of the views. The algorithm passes a reference to a view to each application of an element-access function, ensuring that no two concurrently-executing invocations share the same view. A view can be shared between two applications that do not execute concurrently, but initialization is performed only once per view.

Modifications to the view by the application of element access functions accumulate as partial results. At some point before the algorithm returns, the partial results are combined, two at a time, using the reduction object's combiner operation until a single value remains, which is then assigned back to the live-out object.

T shall meet the requirements of CopyConstructible and MoveAssignable. The expression var = combiner(var, var) shall be well formed.

Template Parameters

- `T`: The value type to be used by the induction object.
- `Op`: The type of the binary function (object) used to perform the reduction operation.

Parameters

- `var`: [in,out] The life-out value to use for the reduction object. This will hold the reduced value after the algorithm is finished executing.
- `identity`: [in] The identity value to use for the reduction operation.
- `combiner`: [in] The binary function (object) used to perform a pairwise reduction on the elements.

Note In order to produce useful results, modifications to the view should be limited to commutative operations closely related to the combiner operation. For example if the combiner is plus<T>, incrementing the view would be consistent with the combiner but doubling it or assigning to it would not.

Return This returns a reduction object of unspecified type having a value type of *T*. When the return value is used by an algorithm, the reference to *var* is used as the live-out object, new views are initialized to a copy of identity, and views are combined by invoking the copy of combiner, passing it the two views to be combined.

namespace 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* = "", *error_code* &*ec* = throwsInstall 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.
- `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.

namespace resource

Typedefs

using hpx::resource::scheduler_function = typedef util::function_nonser< std::unique_p

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.

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::executors::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.

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

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).

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_current = 5
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*)
Get the readable string representing the name of the given thread_state constant.

char **const** ***get_thread_priority_name**(*thread_priority priority*)
Get the readable string representing the name of the given thread_priority constant.

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_name**(std::ptrdiff_t *size*)
Get the readable string representing the given stack size constant.

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_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::size_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).

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.

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.

bool **enumerate_threads** (*util*::function_nonser<bool) thread_id_type
> **const** &*f*, *thread_state_enum* *state* = unknownThe 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.

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, *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, *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, *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, *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.

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)

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.

std::size_t **get_numa_node_number**()

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::executors::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.

threads::*thread_pool_base* ***get_pool**(thread_id_type **const** &*id*, *error_code* &*ec* = throws)

Returns a pointer to the pool that was used to run the current thread

Exceptions

- `If: &ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an hpx::exception with an error code of hpx::yield_aborted if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an hpx::exception with an error code of hpx::null_thread_id. If this function is called while the thread-manager is not running, it will throw an hpx::exception with an error code of hpx::invalid_status.

namespace policies

Enums

enum scheduler_mode

This enumeration describes the possible modes of a scheduler.

Values:

nothing_special = 0

As the name suggests, this option can be used to disable all other options.

do_background_work = 0x1

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 = 0x02

The kernel priority of the os-thread driving the scheduler will be reduced below normal.

delay_exit = 0x04

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 = 0x08

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 = 0x10

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.

default_mode = do_background_work | reduce_thread_priority | delay_exit

This option represents the default mode.

namespace traits

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>::valu
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. 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. 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*>
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. 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. 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. 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. 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*>
void **restore_checkpoint** (*checkpoint* **const** &*c*, T &*t*, Ts&... *ts*)

Resurrect

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).

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.

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.

template <**typename** F, **typename** ... *Ts*>
HPX_HOST_DEVICE util::invoke_result<F, Ts...>::type hpx::util::invoke(F && f, Ts &&...

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*>
HPX_HOST_DEVICE R hpx::util::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.

template <**typename** F, **typename** Tuple>
HPX_HOST_DEVICE detail::invoke_fused_result<F, Tuple>::type hpx::util::invoke_fused(F

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>
HPX_HOST_DEVICE R hpx::util::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.

template <typename Mapper, typename... *T*>
<unspecified> hpx::util::map_pack(Mapper && mapper, T &&... pack)

Maps the pack with the given mapper.

This function tries to visit all plain elements which may be wrapped in:

- homogeneous containers (`std::vector`, `std::list`)
- heterogenous 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.

template <typename Visitor, typename... *T*>
auto **traverse_pack_async**(Visitor &&*visitor*, T&&... *pack*)

Traverses the pack with the given visitor in an asynchronous way.

This function works in the same way as `traverse_pack`, however, we are able to suspend and continue the traversal at later time. Thus we require a visitor callable object which provides three `operator()` overloads as depicted by the code sample below:

```
struct my_async_visitor
{
    template <typename T>
    bool operator()(async_traverse_visit_tag, T&& element)
    {
        return true;
```

```
    }

    template <typename T, typename N>
    void operator()(async_traverse_detach_tag, T&& element, N&& next)
    {
    }

    template <typename T>
    void operator()(async_traverse_complete_tag, T&& pack)
    {
    }
};
```

See `traverse_pack` for a detailed description about the traversal behavior and capabilities.

Return A boost::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 `boost::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 boost::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 `boost::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.

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

file **migrate_from_storage.hpp**

#include <hpx/config.hpp>#include <hpx/lcos/future.hpp>#include <hpx/error_code.hpp>#include <hpx/lcos/detail/future_data.hpp>#include <hpx/lcos/local/detail/condition_variable.hpp>#include <hpx/lcos/local/spinlock.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <hpx/runtime/threads/thread_enums.hpp>#include <hpx/util/steady_clock.hpp>#include <boost/intrusive/slist.hpp>#include <cstddef>#include <mutex>#include <utility>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <type_traits>#include <hpx/runtime/threads/coroutines/detail/get_stack_pointer.hpp>#include <limits>#include <hpx/runtime/threads/thread_executor.hpp>#include <hpx/runtime/get_os_thread_count.hpp>#include <hpx/runtime/threads/cpu_mask.hpp>#include <hpx/util/assert.hpp>#include <climits>#include <cstdint>#include <string>#include <hpx/runtime/threads/policies/scheduler_mode.hpp>#include <hpx/runtime/threads/topology.hpp>#include <hpx/compat/thread.hpp>#include <thread>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/naming_fwd.hpp>#include <hpx/runtime/resource/partitioner_fwd.hpp>#include <hpx/runtime/threads/policies/callback_notifier.hpp>#include <hpx/runtime/threads_fwd.hpp>#include <hpx/util/function.hpp>#include <exception>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include

<memory>#include <hpx/util/spinlock.hpp>#include <hpx/util/itt_notify.hpp>#include <hpx/util/register_locks.hpp>#include <boost/smart_ptr/detail/spinlock.hpp>#include <hpx/util/static.hpp>#include <hpx/compat/mutex.hpp>#include <iosfwd>#include <vector>#include <hwloc.h>#include <hpx/util/atomic_count.hpp>#include <atomic>#include <hpx/util/thread_description.hpp>#include <hpx/util/unique_function.hpp>#include <boost/intrusive_ptr.hpp>#include <chrono>#include <hpx/runtime/threads/thread_helpers.hpp>#include <hpx/throw_exception.hpp>#include <hpx/traits/future_access.hpp>#include <hpx/traits/future_traits.hpp>#include <hpx/traits/is_future.hpp>#include <boost/ref.hpp>#include <functional>#include <hpx/traits/get_remote_result.hpp>#include <hpx/util/annotated_function.hpp>#include <hpx/util/assert_owns_lock.hpp>#include <hpx/traits/has_member_xxx.hpp>#include <hpx/util/bind.hpp>#include <hpx/traits/get_function_address.hpp>#include <hpx/traits/get_function_annotation.hpp>#include <hpx/traits/is_action.hpp>#include <hpx/traits/is_bind_expression.hpp>#include <hpx/traits/is_placeholder.hpp>#include <boost/bind/arg.hpp>#include <hpx/util/decay.hpp>#include <hpx/util/detail/pack.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/invoke_fused.hpp>#include <hpx/util/result_of.hpp>#include <hpx/util/tuple.hpp>#include <hpx/util/void_guard.hpp>#include <hpx/util/one_shot.hpp>#include <hpx/util/unused.hpp>#include <hpx/lcos/detail/future_traits.hpp>#include <hpx/util/always_void.hpp>#include <iterator>#include <hpx/lcos_fwd.hpp>#include <hpx/traits/is_component.hpp>#include <hpx/traits/promise_local_result.hpp>#include <hpx/traits/promise_remote_result.hpp>#include <hpx/runtime/actions/continuation_fwd.hpp>#include <hpx/runtime/serialization/detail/polymorphic_nonintrusive_factory.hpp>
<hpx/runtime/serialization/detail/non_default_constructible.hpp>#include <hpx/traits/needs_automatic_registration.hpp>#inclu
<hpx/traits/polymorphic_traits.hpp>#include <hpx/traits/has_xxx.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <hpx/util/detail/pp/strip_parens.hpp>#include <hpx/util/debug/demangle_helper.hpp>#include <hpx/util/detail/pp/stringize.hpp>#include <hpx/util/jenkins_hash.hpp>#include <typeinfo>#include <unordered_map>#include <hpx/traits/acquire_shared_state.hpp>#include <hpx/util/range.hpp>#include <hpx/traits/detail/reserve.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/traits/is_future_range.hpp>#include <algorithm>#include <hpx/traits/concepts.hpp>#include <hpx/traits/future_then_result.hpp>#include <hpx/util/identity.hpp>#include <hpx/util/lazy_conditional.hpp>#include <hpx/traits/is_executor.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_launch_policy.hpp>#include <hpx/traits/executor_traits.hpp>#include <hpx/util/detected.hpp>#include <hpx/util/allocator_deleter.hpp>#include <hpx/util/internal_allocator.hpp>#include <hpx/util/lazy_enable_if.hpp>#include <hpx/util/serialize_exception.hpp>#include <hpx/lcos/local/packaged_continuation.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/executors/execution_fwd.hpp>#include <hpx/exception_list.hpp>#include <hpx/exception.hpp>#include <boost/system/error_code.hpp>#include <list>#include <hpx/lcos/dataflow.hpp>#include <hpx/lcos/detail/future_transforms.hpp>#include <hpx/traits/acquire_future.hpp>#include <array>#include <hpx/util/deferred_call.hpp>#include <hpx/runtime/get_worker_thread_num.hpp>#include <hpx/traits/extract_action.hpp>#include <hpx/util/pack_traversal_async.hpp>#include <hpx/util/detail/pack_traversal_async_impl.hpp>#include <hpx/util/detail/container_category.hpp>#include <hpx/traits/is_tuple_like.hpp>#include <hpx/parallel/executors/parallel_executor.hpp>#include <hpx/async_launch_policy_dispatch.hpp>#include <hpx/lcos/async_fwd.hpp>#include <hpx/lcos/local/futures_factory.hpp>#include <hpx/lcos/when_all_fwd.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/traits/is_iterator.hpp>#include <boost/iterator/iterator_categories.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <cstdlib>#include <hpx/parallel/executors/post_policy_dispatch.hpp>#include <hpx/parallel/executors/static_chunk_size.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/runtime/serialization/access.hpp>#include <hpx/runtime/serialization/input_archive.hpp>#include <hpx/runtime/serialization/basic_archive.hpp>#include <iostream>#include <map>#include <hpx/runtime/serialization/detail/raw_ptr.hpp>#include <hpx/runtime/serialization/detail/pointer.hpp>#include <hpx/runtime/serialization/detail/polymorphic_id_factory.hpp>#include <hpx/runtime/serialization/detail/polymorphic_intrusive_factory.hpp>#include <hpx/runtime/serialization/string.hpp>#include <hpx/runtime/serialization/input_container.hpp>#include <hpx/runtime/serialization/binary_filter.hpp>#include <hpx/runtime/serialization/container.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/util/spinlock_pool.hpp>#include

*<hpx/util/fibhash.hpp>#include <boost/version.hpp>#include <hpx/runtime/naming/id_type.hpp>#include
<hpx/traits/is_bitwise_serializable.hpp>#include <hpx/util/detail/yield_k.hpp>#include
<hpx/runtime/naming/id_type_impl.hpp>#include <hpx/runtime/serialization/serialization_chunk.hpp>#include
<cstring>#include <hpx/traits/serialization_access_data.hpp>#include <boost/cstdint.hpp>#include
<boost/predef/other/endian.h>#include <hpx/runtime/serialization/output_archive.hpp>#include
<hpx/runtime/serialization/output_container.hpp>#include <hpx/runtime/serialization/detail/polymorphic_nonintrusive_factory_
<hpx/traits/is_executor_parameters.hpp>#include <hpx/parallel/executors/execution_parameters_fwd.hpp>#include
<hpx/lcos/wait_all.hpp>#include <hpx/traits/detail/wrap_int.hpp>#include <hpx/util/bind_back.hpp>#include
<hpx/util/unwrap.hpp>#include <hpx/util/detail/unwrap_impl.hpp>#include
<hpx/util/pack_traversal.hpp>#include <hpx/util/detail/pack_traversal_impl.hpp>#include <stdex-
cept>#include <hpx/components/component_storage/server/migrate_from_storage.hpp>#include
<hpx/runtime/components/runtime_support.hpp>#include <hpx/runtime/applier/applier.hpp>#include
<hpx/runtime/agas_fwd.hpp>#include <hpx/runtime/applier_fwd.hpp>#include
<hpx/runtime/components/component_type.hpp>#include <hpx/traits/component_type_database.hpp>#include
<hpx/util/detail/pp/expand.hpp>#include <hpx/util/detail/pp/nargs.hpp>#include <hpx/util_fwd.hpp>#include
<hpx/runtime/naming/address.hpp>#include <hpx/runtime/parcelset/parcel.hpp>#include
<hpx/runtime/actions_fwd.hpp>#include <hpx/runtime/parcelset_fwd.hpp>#include
<hpx/util/thread_specific_ptr.hpp>#include <boost/thread/tss.hpp>#include <hpx/runtime/components/stubs/runtime_support.hp
<hpx/async.hpp>#include <hpx/lcos/async.hpp>#include <hpx/lcos/detail/async_implementations.hpp>#include
<hpx/lcos/detail/async_implementations_fwd.hpp>#include <hpx/lcos/packaged_action.hpp>#include
<hpx/lcos/promise.hpp>#include <hpx/lcos/detail/promise_base.hpp>#include
<hpx/lcos/detail/promise_lco.hpp>#include <hpx/lcos/base_lco_with_value.hpp>#include
<hpx/lcos/base_lco.hpp>#include <hpx/runtime/actions/basic_action.hpp>#include
<hpx/lcos/sync_fwd.hpp>#include <hpx/runtime/actions/action_support.hpp>#include
<hpx/runtime/components/pinned_ptr.hpp>#include <hpx/runtime/get_lva.hpp>#include
<hpx/runtime/components_fwd.hpp>#include <hpx/traits/managed_component_policies.hpp>#include
<hpx/traits/action_decorate_function.hpp>#include <hpx/traits/component_pin_support.hpp>#include
<hpx/runtime/serialization/base_object.hpp>#include <hpx/runtime/threads/thread_init_data.hpp>#include
<hpx/traits/action_remote_result.hpp>#include <hpx/runtime/actions/basic_action_fwd.hpp>#include
<hpx/runtime/actions/continuation.hpp>#include <hpx/runtime/actions/action_priority.hpp>#include
<hpx/traits/action_priority.hpp>#include <hpx/runtime/actions/trigger.hpp>#include
<hpx/runtime/agas/interface.hpp>#include <boost/dynamic_bitset.hpp>#include
<hpx/runtime/trigger_lco.hpp>#include <hpx/runtime/applier/detail/apply_implementations_fwd.hpp>#include
<hpx/traits/is_continuation.hpp>#include <hpx/util/logging.hpp>#include <hpx/runtime/actions/detail/action_factory.hpp>#incl
<hpx/runtime/actions/detail/invocation_count_registry.hpp>#include <hpx/performance_counters/counters_fwd.hpp>#include
<hpx/runtime/actions/preassigned_action_id.hpp>#include <hpx/runtime/actions/transfer_action.hpp>#include
<hpx/runtime/actions/transfer_base_action.hpp>#include <hpx/runtime/actions/base_action.hpp>#include
<hpx/runtime/threads/thread_id_type.hpp>#include <hpx/runtime/serialization/unique_ptr.hpp>#include
<hpx/traits/action_does_termination_detection.hpp>#include <hpx/traits/action_message_handler.hpp>#include
<hpx/traits/action_schedule_thread.hpp>#include <hpx/traits/action_serialization_filter.hpp>#include
<hpx/traits/action_stacksize.hpp>#include <hpx/traits/action_was_object_migrated.hpp>#include
<hpx/util/get_and_reset_value.hpp>#include <hpx/runtime/applier/apply_helper.hpp>#include
<hpx/runtime_fwd.hpp>#include <hpx/runtime/basename_registration_fwd.hpp>#include
<hpx/components_fwd.hpp>#include <hpx/runtime/components/make_client.hpp>#include
<hpx/traits/is_client.hpp>#include <hpx/runtime/config_entry.hpp>#include
<hpx/runtime/find_localities.hpp>#include <hpx/runtime/get_colocation_id.hpp>#include
<hpx/runtime/get_locality_id.hpp>#include <hpx/runtime/get_locality_name.hpp>#include
<hpx/runtime/get_num_localities.hpp>#include <hpx/runtime/get_thread_name.hpp>#include
<hpx/runtime/report_error.hpp>#include <hpx/runtime/runtime_fwd.hpp>#include
<hpx/runtime/runtime_mode.hpp>#include <hpx/runtime/set_parcel_write_handler.hpp>#include
<hpx/runtime/shutdown_function.hpp>#include <hpx/runtime/startup_function.hpp>#include
<hpx/state.hpp>#include <hpx/traits/action_continuation.hpp>#include <hpx/traits/action_decorate_continuation.hpp>#include
<hpx/traits/action_select_direct_execution.hpp>#include <hpx/runtime/parcelset/detail/per_action_data_counter_registry.hpp>#
<hpx/runtime/actions/transfer_continuation_action.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include*

<boost/utility/string_ref.hpp>#include <sstream>#include <hpx/runtime/actions/component_action.hpp>#include <hpx/runtime/components/server/managed_component_base.hpp>#include <hpx/runtime/components/server/create_component_ <hpx/runtime/components/server/component_heap.hpp>#include <hpx/util/reinitializable_static.hpp>#include <hpx/util/bind_front.hpp>#include <hpx/util/static_reinit.hpp>#include <hpx/runtime/components/server/wrapper_heap.hpp>#i <hpx/util/generate_unique_ids.hpp>#include <hpx/util/wrapper_heap_base.hpp>#include <new>#include <hpx/runtime/components/server/wrapper_heap_list.hpp>#include <hpx/util/one_size_heap_list.hpp>#include <hpx/util/unlock_guard.hpp>#include <hpx/plugins/parcel/coalescing_message_handler_registration.hpp>#include <hpx/runtime/components/server/component_base <hpx/util/ini.hpp>#include <boost/lexical_cast.hpp>#include <hpx/lcos/local/promise.hpp>#include <boost/utility/swap.hpp>#include <hpx/runtime/applier/apply.hpp>#include <hpx/runtime/applier/detail/apply_implementations.hpp>#include <hpx/traits/action_is_target_valid.hpp>#include <hpx/traits/component_supports_migration.hpp>#include <hpx/util/format.hpp>#include <cctype>#include <cstdio>#include <ostream>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/lcos/detail/async_colocated_fwd.hpp>#include <hpx/runtime/naming/unmanaged.hpp>#include <hpx/runtime/parcelset/detail/parcel_await.hpp>#include <hpx/runtime/parcelset/put_parcel.hpp>#include <hpx/runtime.hpp>#include <hpx/performance_counters/counters.hpp>#include <hpx/runtime/parcelset/locality.hpp>#include <hpx/runtime/serialization/map.hpp>#include <hpx/runtime/thread_hooks.hpp>#include <hpx/util/runtime_configuration.hpp>#include <hpx/runtime/components/static_factory_data.hpp>#include <hpx/util/plugin/export_plugin.hpp>#include <hpx/util/plugin/abstract_factory.hpp>#include <hpx/util/plugin/virtual_constructor.hpp>#include <hpx/util/plugin/config.hpp>#include <boost/any.hpp>#include <boost/shared_ptr.hpp>#include <hpx/util/plugin/concrete_factory.hpp>#include <hpx/util/plugin/plugin_wrapper.hpp>#include <boost/algorithm/string/case_conv.hpp>#include <hpx/util/plugin/dll.hpp>#include <hpx/util/plugin/detail/dll_dlopen.hpp>#include <boost/filesystem/convenience.hpp>#include <boost/filesystem/path.hpp>#include <link.h>#include <dlfcn.h>#include <limits.h>#include <hpx/plugins/plugin_registry_base.hpp>#include <hpx/util/plugin.hpp>#include <hpx/util/plugin/plugin_factory.hpp>#include <boost/filesystem.hpp>#include <set>#include <hpx/runtime/naming/split_gid.hpp>#include <hpx/runtime/parcelset/parcelhandler.hpp>#include <hpx/runtime/parcelset/parcelport.hpp>#include <hpx/performance_counters/parcels/data_point.hpp>#include <hpx/performance_counters/parcels/gatherer.hpp>#include <hpx/lcos/local/no_mutex.hpp>#include <hpx/runtime/parcelset/detail/per_action_data_counter.hpp>#include <hpx/util/high_resolution_timer.hpp>#include <hpx/util/high_resolution_clock.hpp>#include <hpx/plugins/parcelport_factory_base.hpp>#include <hpx/traits/component_type_is_compatible.hpp>#include <hpx/traits/is_valid_action.hpp>#include <hpx/runtime/applier/apply_callback.hpp>#include <boost/asio/error.hpp>#include <hpx/runtime/threads/thread.hpp>#include <hpx/lcos/sync.hpp>#include <hpx/lcos/detail/sync_implementations.hpp>#include <hpx/lcos/detail/sync_implementations_fwd.hpp>#include <hpx/lcos/async_continue.hpp>#include <hpx/lcos/async_continue_fwd.hpp>#include <hpx/util/bind_action.hpp>#include <hpx/runtime/actions/manage_object_action. <hpx/runtime/serialization/array.hpp>#include <boost/array.hpp>#include <hpx/runtime/serialization/serialize_buffer.hpp>#in <hpx/traits/supports_streaming_with_any.hpp>#include <boost/shared_array.hpp>#include <hpx/runtime/components/server/runtime_support.hpp>#include <hpx/compat/condition_variable.hpp>#include <condition_variable>#include <hpx/lcos/local/condition_variable.hpp>#include <hpx/lcos/local/mutex.hpp>#include <hpx/plugins/plugin_factory_base.hpp>#include <hpx/runtime/components/server/create_component.hpp>#include <hpx/runtime/find_here.hpp>#include <boost/program_options/options_description.hpp>#include <hpx/runtime/serialization/vector.hpp>#include <hpx/runtime/serialization/detail/serialize_collection.hpp>#include <hpx/runtime/components/server/migrate_component.hpp>#include <hpx/runtime/actions/plain_action.hpp>#include <hpx/runtime/get_ptr.hpp>#include <hpx/runtime/agas/gva.hpp>#include <boost/io/ios_state.hpp>#include <hpx/components/component_storage/export_definitions.hpp>#include <hpx/config/export_definitions.hpp>#include <hpx/components/component_storage/server/component_storage.hpp>#include <hpx/components/containers/unordered/unorder <hpx/runtime/components/copy_component.hpp>#include <hpx/lcos/detail/async_colocated.hpp>#include <hpx/runtime/agas/primary_namespace.hpp>#include <hpx/runtime/agas/server/primary_namespace.hpp>#include <hpx/runtime/components/server/fixed_component_base.hpp>#include <hpx/runtime/applier/bind_naming_wrappers.hpp>#incl <hpx/util/functional/colocated_helpers.hpp>#include <hpx/runtime/components/server/copy_component.hpp>#include <hpx/runtime/components/new.hpp>#include <hpx/runtime/components/default_distribution_policy.hpp>#include <hpx/runtime/serialization/shared_ptr.hpp>#include <hpx/runtime/components/server/distributed_metadata_base.hpp>#include

<hpx/runtime/components/server/simple_component_base.hpp>#include <hpx/runtime/components/server/component.hpp>#inc <hpx/runtime/serialization/unordered_map.hpp>#include <hpx/components/containers/container_distribution_policy.hpp>#incl <hpx/components/containers/unordered/partition_unordered_map_component.hpp>#include <hpx/lcos/reduce.hpp>#include <hpx/runtime/components/component_factory.hpp>#include <hpx/runtime/components/server/locking_hook.hpp>#include <hpx/runtime/threads/coroutines/coroutine.hpp>#include <hpx/runtime/threads/coroutines/coroutine_fwd.hpp>#include <hpx/runtime/threads/coroutines/detail/coroutine_accessor.hpp># <hpx/runtime/threads/coroutines/detail/coroutine_impl.hpp>#include <hpx/runtime/threads/coroutines/detail/context_base.hpp> <hpx/runtime/threads/coroutines/detail/context_impl.hpp>#include <hpx/runtime/threads/coroutines/detail/swap_context.hpp># <hpx/runtime/threads/coroutines/detail/tss.hpp>#include <hpx/runtime/threads/coroutines/exception.hpp>#include <hpx/runtime/threads/coroutines/detail/coroutine_self.hpp>#include <tuple>#include <hpx/components/containers/unordered/u <hpx/util/iterator_adaptor.hpp>#include <hpx/util/iterator_facade.hpp>#include <boost/integer.hpp>

file **`migrate_to_storage.hpp`**

#include <hpx/config.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/traits/is_component.hpp>#include <hpx/components/component_storage/component_storage.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/naming/address.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/components/component_storage/server/component_storage.hpp>#include <cstddef>#include <vector>#include <hpx/components/component_storage/server/migrate_to_storage.hpp>#include <hpx/throw_exception.hpp>#include <hpx/util/bind_back.hpp>#include <hpx/components/component_storage/export_definition <cstdint>#include <memory>#include <utility>#include <type_traits>

file **`error.hpp`**

#include <hpx/config.hpp>#include <boost/system/error_code.hpp>#include <string>

file **`error_code.hpp`**

#include <hpx/config.hpp>#include <hpx/error.hpp>#include <hpx/exception_fwd.hpp>#include <boost/system/error_code.hpp>#include <exception>#include <stdexcept>#include <string>#include <hpx/throw_exception.hpp>

file **`exception.hpp`**

#include <hpx/config.hpp>#include <hpx/error.hpp>#include <hpx/error_code.hpp>#include <hpx/error.hpp>#include <hpx/exception_fwd.hpp>#include <boost/system/error_code.hpp>#include <exception>#include <stdexcept>#include <string>#include <hpx/throw_exception.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/exception_info.hpp>#include <hpx/error_code.hpp>#include <hpx/util/detail/pack.hpp>#include <cstddef>#include <type_traits>#include <hpx/util/tuple.hpp>#include <hpx/runtime/serialization/detail/non_default_constructible.hpp>#include <memory>#include <hpx/traits/is_bitwise_serializable.hpp>#include <hpx/util/decay.hpp>#include <boost/ref.hpp>#include </hpx/build/docs/hpx/util/functional>#include <utility>#include <boost/array.hpp>#include <array>#include <algorithm>#include <typeinfo>#include <hpx/runtime/naming_fwd.hpp>#include <hpx/runtime/agas_fwd.hpp>#include <hpx/util/function.hpp>#include <cstdint>#include <boost/system/system_error.hpp>#include <hpx/config/warnings_prefix.hpp>#include <hpx/throw_exception.hpp>#include <hpx/config/warnings_suffix.hpp>

file **`exception_fwd.hpp`**

#include <hpx/config.hpp>#include <hpx/error.hpp>#include <hpx/throw_exception.hpp>

file **`exception_list.hpp`**

#include <hpx/config.hpp>#include <hpx/exception.hpp>#include <hpx/error.hpp>#include <hpx/error_code.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/exception_info.hpp>#include <hpx/runtime/naming_fwd.hpp>#include <boost/system/error_code.hpp>#include <boost/system/system_error.hpp>#include <cstddef>#include <cstdint>#include <exception>#include <string>#include <hpx/config/warnings_prefix.hpp>#include <hpx/throw_exception.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/lcos/local/spinlock.hpp>#include <hpx/runtime/threads/thread_helpers.hpp>#include <hpx/runtime/threads_fwd.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <hpx/runtime/threads/coroutines/coroutine_fwd.hpp>#include <hpx/runtime/threads/thread_enums.hpp>#include

<hpx/runtime/threads/detail/combined_tagged_state.hpp>#include <hpx/util/assert.hpp>#include <hpx/runtime/threads/thread_id_type.hpp>#include <hpx/config/constexpr.hpp>#include <hpx/config/export_definitions.hpp>#include <functional>#include <iosfwd>#include <hpx/util_fwd.hpp>#include <hpx/util/function.hpp>#include <hpx/util/unique_function.hpp>#include <utility>#include <memory>#include <hpx/runtime/thread_pool_helpers.hpp>#include <hpx/runtime/threads/policies/scheduler_mode.hpp>#include <hpx/util/register_locks.hpp>#include <hpx/traits/has_member_xxx.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <type_traits>#include <hpx/util/steady_clock.hpp>#include <chrono>#include <hpx/util/thread_description.hpp>#include <hpx/runtime/actions/basic_action_fwd.hpp>#include <hpx/runtime/actions/preassigned_action_id.hpp>#include <hpx/traits/get_function_address.hpp>#include <hpx/traits/get_function_annotation.hpp>#include <hpx/traits/is_action.hpp>#include <hpx/util/always_void.hpp>#include <hpx/util/decay.hpp>#include <atomic>#include <hpx/util/detail/yield_k.hpp>#include <sched.h>#include <time.h>#include <hpx/util/itt_notify.hpp>#include <boost/smart_ptr/detail/spinlock.hpp>#include <list>#include <mutex>

file **hpx_finalize.hpp**

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

file **hpx_init.hpp**

#include <hpx/config.hpp>#include <hpx/config/attributes.hpp>#include <hpx/config/defines.hpp>#include <hpx/config/compiler_specific.hpp>#include <hpx/config/branch_hints.hpp>#include <hpx/config/compiler_fence.hpp>#include <hpx/config/compiler_native_tls.hpp>#include <ciso646>#include <hpx/config/constexpr.hpp>#include <hpx/config/debug.hpp>#include <hpx/config/emulate_deleted.hpp>#include <hpx/config/export_definitions.hpp>#include <hpx/config/forceinline.hpp>#include <hpx/config/lambda_capture.hpp>#include <utility>#include <hpx/config/manual_profiling.hpp>#include <hpx/config/threads_stack.hpp>#include <hpx/config/version.hpp>#include <hpx/config/weak_symbol.hpp>#include <boost/version.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <hpx/util/detail/pp/config.hpp>#include <hpx/util/detail/pp/stringize.hpp>#include <hpx/hpx_finalize.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/error.hpp>#include <boost/system/error_code.hpp>#include <string>#include <hpx/throw_exception.hpp>#include <hpx/util/detail/pp/expand.hpp>#include <hpx/util/detail/pp/nargs.hpp>#include <boost/current_function.hpp>#include <exception>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/hpx_suspend.hpp>#include <hpx/runtime/runtime_mode.hpp>#include <hpx/runtime/shutdown_function.hpp>#include <hpx/util/unique_function.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <hpx/util/detail/pp/strip_parens.hpp>#include <type_traits>#include <hpx/traits/get_function_address.hpp>#include <cstddef>#include <memory>#include <hpx/traits/get_function_annotation.hpp>#include <hpx/util/itt_notify.hpp>#include <cstdint>#include <cstring>#include <hpx/traits/is_callable.hpp>#include <hpx/util/always_void.hpp>#include <hpx/util/result_of.hpp>#include <boost/ref.hpp>#include <hpx/util/detail/basic_function.hpp>#include <hpx/util/detail/empty_function.hpp>#include <hpx/util/detail/function_registration.hpp>#include <hpx/util/debug/demangle_helper.hpp>#include <typeinfo>#include <cstdlib>#include <hpx/util/detail/vtable/vtable.hpp>#include <hpx/util/assert.hpp>#include <assert.h>#include <iostream>#include <hpx/util/detail/vtable/serializable_function_vtable.hpp>#include <hpx/runtime/serialization/detail/polymorphic_intrusive_factory.hpp>#include <hpx/util/jenkins_hash.hpp>#include <random>#include <unordered_map>#include <hpx/util/detail/vtable/serializable_vtable.hpp>#include <hpx/util/detail/vtable/unique_function_vtable.hpp>#include <hpx/util/detail/vtable/callable_vtable.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/void_guard.hpp>#include </hpx/build/docs/hpx/util/functional>#include <hpx/util_fwd.hpp>#include <hpx/runtime/startup_function.hpp>#include <hpx/util/function.hpp>#include <hpx/util/detail/vtable/function_vtable.hpp>#include <hpx/util/detail/vtable/copyable_vtable.hpp>#include <boost/program_options/options_description.hpp>#include <boost/program_options/variables_map.hpp>#include <vector>

file **hpx_start.hpp**

#include <hpx/config.hpp>#include <hpx/hpx_finalize.hpp>#include <hpx/runtime/runtime_mode.hpp>#include <hpx/runtime/shutdown_function.hpp>#include <hpx/runtime/startup_function.hpp>#include <hpx/util/function.hpp>#include <boost/program_options/options_description.hpp>#include <boost/program_options/variables_map.hpp>#include <cstddef>#include <string>#include <vector>

file **hpx_suspend.hpp**

#include <hpx/exception_fwd.hpp>

file **barrier.hpp**

#include <hpx/config.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/components/server/managed_component_base <hpx/runtime/launch_policy.hpp>#include <boost/intrusive_ptr.hpp>#include <cstddef>#include <string>#include <utility>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>

file **broadcast.hpp**

file **fold.hpp**

file **gather.hpp**

Defines

HPX_REGISTER_GATHER_DECLARATION(type, name)

Declare a gather object named *name* for a given data type *type*.

The macro *HPX_REGISTER_GATHER_DECLARATION* can be used to declare all facilities necessary for a (possibly remote) gather operation.

The parameter *type* specifies for which data type the gather operations should be enabled.

The (optional) parameter *name* should be a unique C-style identifier which will be internally used to identify a particular gather operation. If this defaults to *<type>_gather* if not specified.

Note The macro *HPX_REGISTER_GATHER_DECLARATION* can be used with 1 or 2 arguments. The second argument is optional and defaults to *<type>_gather*.

HPX_REGISTER_GATHER(type, name)

Define a gather object named *name* for a given data type *type*.

The macro *HPX_REGISTER_GATHER* can be used to define all facilities necessary for a (possibly remote) gather operation.

The parameter *type* specifies for which data type the gather operations should be enabled.

The (optional) parameter *name* should be a unique C-style identifier which will be internally used to identify a particular gather operation. If this defaults to *<type>_gather* if not specified.

Note The macro *HPX_REGISTER_GATHER* can be used with 1 or 2 arguments. The second argument is optional and defaults to *<type>_gather*.

file **split_future.hpp**

file **wait_all.hpp**

file **wait_any.hpp**

file **wait_each.hpp**

file **wait_some.hpp**

file **when_all.hpp**

file **when_any.hpp**

file **when_each.hpp**

file **when_some.hpp**

file **algorithm.hpp**

#include <hpx/config.hpp>#include <algorithm>#include <hpx/parallel/algorithms/adjacent_find.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <hpx/throw_exception.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <hpx/util/decay.hpp>#include <type_traits>#include <utility>#include <hpx/parallel/exception_list.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/datapar/execution_policy.hpp>#include <hpx/parallel/execution_policy_fwd.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/executors/execution_parameters.hpp>#include <hpx/parallel/executors/parallel_executor.hpp>#include <hpx/parallel/executors/rebind_executor.hpp>#include <hpx/parallel/executors/sequenced_executor.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/traits/executor_traits.hpp>#include <hpx/traits/is_execution_policy.hpp>#include <hpx/traits/is_executor.hpp>#include <hpx/traits/is_executor_parameters.hpp>#include <hpx/traits/is_launch_policy.hpp>#include <memory>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/util/detail/scoped_executor_parameters.hpp>#include <hpx/util/tuple.hpp>#include <string>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/cancellation_token.hpp>#include <atomic>#include <functional>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/result_of.hpp>#include <cstddef>#include <iterator>#include <vector>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/dataflow.hpp>#include <hpx/lcos/dataflow.hpp>#include <hpx/exception_list.hpp>#include <hpx/lcos/wait_all.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/util/detail/chunk_size.hpp>#include <hpx/util/iterator_range.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/executors/execution_information.hpp>#include <hpx/runtime/threads/topology.hpp>#include <hpx/traits/detail/wrap_int.hpp>#include <hpx/parallel/executors/execution_information_fwd.hpp>#include <hpx/parallel/executors/execution_fwd.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <hpx/parallel/util/detail/chunk_size_iterator.hpp>#include <hpx/util/iterator_facade.hpp>#include <hpx/parallel/util/detail/handle_local_exceptions.hpp>#include <hpx/async.hpp>#include <hpx/hpx_finalize.hpp>#include <exception>#include <list>#include <hpx/parallel/util/detail/partitioner_iteration.hpp>#incluc <hpx/util/invoke_fused.hpp>#include <hpx/parallel/util/detail/select_partitioner.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/tagged.hpp>#include <hpx/util/detail/pack.hpp>#include </hpx/build/docs/hpx/util/functional>#include <hpx/util/zip_iterator.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/util/functional/segmented_iterator_helpers.hpp>#include <hpx/parallel/algorithms/all_any_none.hpp>#include <hpx/util/void_guard.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/vector_pack_load_store.hpp>#include <hpx/parallel/traits/vector_pack_type.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/util/always_void.hpp>#include <hpx/parallel/util/invoke_projected.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/transfer.hpp>#include <hpx/parallel/segmented_algorithms/detail/transfer.hpp>#include <hpx/parallel/segmented_algorithms/detail/dispatch.hpp>#inc <hpx/runtime/actions/plain_action.hpp>#include <hpx/runtime/components/colocating_distribution_policy.hpp>#include <hpx/lcos/detail/async_colocated.hpp>#include <hpx/lcos/detail/async_colocated_callback.hpp>#include <hpx/runtime/agas/primary_namespace.hpp>#include <hpx/runtime/agas/server/primary_namespace.hpp>#include <hpx/lcos/async_continue_callback.hpp>#include <hpx/lcos/async_callback_fwd.hpp>#include <hpx/lcos/async_fwd.hpp>#include <hpx/runtime/actions/basic_action_fwd.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/lcos/async_continue.hpp>#include <hpx/runtime/applier/apply_callback.hpp>#include <hpx/traits/extract_action.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <hpx/traits/promise_local_result.hpp>#include <hpx/traits/promise_remote_result.hpp>#include <hpx/lcos/detail/async_colocated_callback_fwd.hpp>#include

<hpx/lcos/detail/async_colocated_fwd.hpp>#include <hpx/lcos/detail/async_implementations.hpp>#include <hpx/runtime/applier/detail/apply_colocated_callback_fwd.hpp>#include <hpx/runtime/actions/action_support.hpp>#include <hpx/runtime/applier/detail/apply_colocated_fwd.hpp>#include <hpx/traits/is_continuation.hpp>#include <hpx/runtime/applier/detail/apply_implementations.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/name.hpp>#include <hpx/parallel/util/detail/handle_remote_exceptions.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/parallel/util/transfer.hpp>#include <hpx/traits/pointer_category.hpp>#include <cstring>#include <boost/shared_array.hpp>#include <hpx/parallel/algorithms/count.hpp>#include <hpx/util/bind_back.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/traits/vector_pack_count_bits.hpp>#include <hpx/parallel/algorithms/equal.hpp>#include <hpx/parallel/algorithms/fill.hpp>#include <hpx/traits/is_value_proxy.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/util/annotated_function.hpp>#include <hpx/util/identity.hpp>#include <cstdint>#include <hpx/parallel/algorithms/find.hpp>#include <hpx/parallel/util/compare_projected.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/algorithms/generate.hpp>#include <hpx/parallel/algorithms/includes.hpp>#include <hpx/parallel/algorithms/is_heap.hpp>#include <hpx/parallel/algorithms/is_partitioned.hpp>#include <hpx/parallel/algorithms/is_sorted.hpp>#include <hpx/parallel/algorithms/lexicographical_compare.hpp>#include <hpx/parallel/algorithms/mismatch.hpp>#include <hpx/parallel/algorithms/merge.hpp>#include <hpx/util/tagged_tuple.hpp>#include <hpx/parallel/algorithms/minmax.hpp>#include <hpx/parallel/algorithms/mismatch.hpp>#include <hpx/parallel/algorithms/move.hpp>#include <hpx/parallel/algorithms/partition.hpp>#include <hpx/lcos/local/spinlock.hpp>#include <hpx/parallel/algorithms/remove.hpp>#include <hpx/parallel/algorithms/remove_copy.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/replace.hpp>#include <hpx/parallel/algorithms/reverse.hpp>#include <hpx/parallel/algorithms/rotate.hpp>#include <hpx/parallel/algorithms/reverse.hpp>#include <hpx/parallel/algorithms/search.hpp>#include <hpx/parallel/algorithms/set_difference.hpp>#include <hpx/parallel/algorithms/detail/set_operation.hpp>#include <hpx/parallel/algorithms/set_intersection.hpp>#include <hpx/parallel/algorithms/set_symmetric_difference.hpp>#include <hpx/parallel/algorithms/set_union.hpp>#include <hpx/parallel/algorithms/sort.hpp>#include <hpx/parallel/algorithms/swap_ranges.hpp>#include <hpx/parallel/algorithms/unique.hpp>#include <hpx/parallel/algorithms/for_loop.hpp>#include <hpx/parallel/algorithms/for_loop_induction.hpp>#include <cstdlib>#include <hpx/parallel/algorithms/for_loop_reduction.hpp>#include <hpx/runtime/get_os_thread_count.hpp>#include <hpx/runtime/get_worker_thread_num.hpp>

file **adjacent_difference.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/zip_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>

file **adjacent_find.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **all_any_none.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/range.hpp>#include <hpx/util/void_guard.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/invoke_projected.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include

<hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **all_any_none.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/all_any_none.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/util/decay.hpp>#include <hpx/util/result_of.hpp>#include <hpx/parallel/traits/projected.hpp>#include <iterator>#include <type_traits>#include <hpx/parallel/util/projection_identity.hpp>#include <utility>

file **copy.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/algorithms/detail/transfer.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/parallel/util/transfer.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <cstring>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>#include <boost/shared_array.hpp>

file **copy.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <type_traits>#include <utility>

file **count.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <hpx/util/bind_back.hpp>#include <hpx/util/range.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/vector_pack_count_bits.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/invoke_projected.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <functional>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **count.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/count.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **destroy.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/void_guard.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>

file **equal.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include

<hpx/parallel/util/zip_iterator.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **exclusive_scan.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/util/zip_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/inclusive_scan.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>#include <hpx/parallel/execution_policy.hpp>

file **fill.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_value_proxy.hpp>#include <hpx/util/void_guard.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>

file **fill.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_execution_policy.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/fill.hpp>#include <type_traits>#include <utility>

file **find.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/compare_projected.hpp>#include <hpx/parallel/util/invoke_projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **find.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_execution_policy.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/find.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **for_each.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_value_proxy.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <hpx/util/annotated_function.hpp>#include <hpx/util/identity.hpp>#include <hpx/util/invoke.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <algorithm>#include <cstddef>#include <cstdint>#include <iterator>#include <type_traits>#include <utility>

file **for_each.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include

<type_traits>#include <utility>

file **for_loop.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/annotated_function.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/decay.hpp>#include <hpx/util/detail/pack.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tuple.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/algorithms/for_loop_induction.hpp>#include <hpx/parallel/algorithms/for_loop_reduction.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <cstdint>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **for_loop_induction.hpp**

#include <hpx/config.hpp>#include <hpx/util/decay.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <cstddef>#include <cstdlib>#include <type_traits>#include <utility>

file **for_loop_reduction.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/get_os_thread_count.hpp>#include <hpx/runtime/get_worker_thread_num.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/decay.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <boost/shared_array.hpp>#include <cstddef>#include <cstdlib>#include <functional>#include <type_traits>#include <utility>

file **generate.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>

file **generate.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/generate.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **includes.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/cancellation_token.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **inclusive_scan.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/util/zip_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>

file **is_heap.hpp**

#include <hpx/config.hpp>#include <hpx/async.hpp>#include <hpx/lcos/future.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include

<hpx/parallel/execution_policy.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <list>#include <vector>#include <type_traits>#include <utility>

file **is_heap.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/is_heap.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **is_partitioned.hpp**

#include <hpx/config.hpp>#include <hpx/lcos/future.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/cancellation_token.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <functional>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **is_sorted.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/cancellation_token.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <functional>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **lexicographical_compare.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/algorithms/mismatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **merge.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_tuple.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/algorithms/detail/transfer.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/compare_projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/detail/handle_local_exceptions.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/transfer.hpp>#include <algorithm>#include <cstddef>#include <exception>#include <iterator>#include <list>#include <memory>#include <type_traits>#include <utility>#include <vector>

file **merge.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_tuple.hpp>#include <hpx/parallel/algorithms/merge.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <type_traits>#include <utility>

file **minmax.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/compare_projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file `minmax.hpp`

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/minmax.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <type_traits>#include <utility>

file `mismatch.hpp`

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file `move.hpp`

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/transfer.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/transfer.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>

file `partition.hpp`

#include <hpx/config.hpp>#include <hpx/async.hpp>#include <hpx/lcos/dataflow.hpp>#include <hpx/lcos/future.hpp>#include <hpx/lcos/local/spinlock.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_tuple.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/exception_list.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/executors/execution_information.hpp>#include <hpx/parallel/executors/execution_parameters.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/detail/handle_local_exceptions.hpp>#include <hpx/parallel/util/invoke_projected.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <cstdint>#include <exception>#include <iterator>#include <list>#include <type_traits>#include <utility>#include <vector>#include <boost/shared_array.hpp>

file `partition.hpp`

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_tuple.hpp>#include <hpx/parallel/algorithms/partition.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <type_traits>#include <utility>

file `reduce.hpp`

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/range.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include

<hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>

file **reduce.hpp**

file **reduce_by_key.hpp**

#include <hpx/config.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/algorithms/inclusive_scan.hpp>#include <hpx/parallel/algorithms/sort.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <hpx/util/range.hpp>#include <hpx/util/transform_iterator.hpp>#include <hpx/util/result_of.hpp>#include <hpx/util/iterator_adaptor.hpp>#include <iterator>#include <type_traits>#include <hpx/util/tuple.hpp>#include <cstdint>#include <functional>#include <utility>#include <vector>

file **remove.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/algorithms/detail/transfer.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/invoke_projected.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/parallel/util/transfer.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <cstring>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>#include <boost/shared_array.hpp>

file **remove.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/remove.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <type_traits>#include <utility>

file **remove_copy.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **remove_copy.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/remove_copy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **replace.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include

<hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **replace.hpp**
#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/replace.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **reverse.hpp**
#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **reverse.hpp**
#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/reverse.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **rotate.hpp**
#include <hpx/config.hpp>#include <hpx/dataflow.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/reverse.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/transfer.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **rotate.hpp**
#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/rotate.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **search.hpp**
#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/compare_projected.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **search.hpp**
#include <hpx/config.hpp>#include <hpx/traits/is_execution_policy.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/search.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <cstddef>#include <type_traits>#include <utility>

file **set_difference.hpp**
#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/decay.hpp>#include

<hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/set_operation.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **set_intersection.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/decay.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/set_operation.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **set_symmetric_difference.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/decay.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/set_operation.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **set_union.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/decay.hpp>#include <hpx/parallel/algorithms/copy.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/set_operation.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **sort.hpp**

#include <hpx/config.hpp>#include <hpx/dataflow.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/decay.hpp>#include <hpx/util/invoke.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/exception_list.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/compare_projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <algorithm>#include <cstddef>#include <exception>#include <functional>#include <iterator>#include <list>#include <type_traits>#include <utility>

file **sort.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/parallel/algorithms/sort.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <type_traits>#include <utility>

file **sort_by_key.hpp**

#include <hpx/config.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/tuple.hpp>#include <hpx/parallel/algorithms/sort.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **swap_ranges.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/for_each.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <iterator>#include <type_traits>#include <utility>

file **transform.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include

<hpx/util/annotated_function.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/tagged_tuple.hpp>#include <hpx/util/tuple.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/transform_loop.hpp>#include <hpx/parallel/util/cancellation_token.hpp>#include <hpx/traits/is_execution_policy.hpp>#include <hpx/util/invoke.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <hpx/parallel/util/zip_iterator.hpp>#include <cstdint>

file **`transform.hpp`**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/tagged_tuple.hpp>#include <hpx/parallel/algorithms/transform.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <hpx/util/annotated_function.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tuple.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/transform_loop.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <cstdint>#include <iterator>#include <type_traits>#include <utility>#include <hpx/parallel/traits/projected_range.hpp>

file **`transform_exclusive_scan.hpp`**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/result_of.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/transform_inclusive_scan.hpp>#include <hpx/util/invoke.hpp>#include <hpx/parallel/algorithms/inclusive_scan.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>#include <hpx/parallel/execution_policy.hpp>

file **`transform_inclusive_scan.hpp`**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/result_of.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/inclusive_scan.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>

file **`transform_reduce.hpp`**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/segmented_iterator_traits.hpp>#include <hpx/util/range.hpp>#include <hpx/util/result_of.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>

file **`transform_reduce_binary.hpp`**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_callable.hpp>#include

<hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/result_of.hpp>#include <hpx/util/zip_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <hpx/util/unused.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <numeric>#include <type_traits>#include <utility>#include <vector>

file **uninitialized_copy.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner_with_cleanup.hpp>#include <hpx/dataflow.hpp>#include <hpx/exception_list.hpp>#include <hpx/lcos/wait_all.hpp>#include <hpx/util/unused.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/executors/execution_parameters.hpp>#include <hpx/parallel/util/detail/chunk_size.hpp>#include <hpx/parallel/util/detail/handle_local_exceptions.hpp>#include <hpx/parallel/util/detail/partitioner_iteration.hpp>#include <hpx/parallel/util/detail/scoped_executor_parameters.hpp>#include <hpx/parallel/util/detail/select_partitioner.hpp>#include <hpx/parallel/util/partitioner.hpp>#include <algorithm>#include <cstddef>#include <exception>#include <list>#include <memory>#include <type_traits>#include <utility>#include <vector>#include <hpx/parallel/util/zip_iterator.hpp>#include <iterator>

file **uninitialized_default_construct.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/void_guard.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner_with_cleanup.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>

file **uninitialized_fill.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner_with_cleanup.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>

file **uninitialized_move.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner_with_cleanup.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>

file **uninitialized_value_construct.hpp**

#include <hpx/config.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/void_guard.hpp>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/partitioner_with_cleanup.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>

file **unique.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/util/unused.hpp>#include

<hpx/parallel/algorithms/detail/dispatch.hpp>#include <hpx/parallel/algorithms/detail/is_negative.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/algorithms/detail/transfer.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/util/compare_projected.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <hpx/parallel/util/foreach_partitioner.hpp>#include <hpx/parallel/util/loop.hpp>#include <hpx/parallel/util/projection_identity.hpp>#include <hpx/parallel/util/scan_partitioner.hpp>#include <hpx/parallel/util/transfer.hpp>#include <hpx/parallel/util/zip_iterator.hpp>#include <algorithm>#include <cstddef>#include <cstring>#include <iterator>#include <memory>#include <type_traits>#include <utility>#include <vector>#include <boost/shared_array.hpp>

file **unique.hpp**

#include <hpx/config.hpp>#include <hpx/traits/concepts.hpp>#include <hpx/traits/is_iterator.hpp>#include <hpx/traits/is_range.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tagged_pair.hpp>#include <hpx/parallel/algorithms/unique.hpp>#include <hpx/parallel/tagspec.hpp>#include <hpx/parallel/traits/projected.hpp>#include <hpx/parallel/traits/projected_range.hpp>#include <type_traits>#include <utility>

file **execution_policy.hpp**

#include <hpx/config.hpp>#include <hpx/parallel/datapar/execution_policy.hpp>#include <hpx/parallel/execution_policy_fwd.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/executors/execution_parameters.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/serialization/base_object.hpp>#include <hpx/traits/detail/wrap_int.hpp>#include <hpx/traits/has_member_xxx.hpp>#include <hpx/traits/is_executor.hpp>#include <hpx/traits/is_executor_parameters.hpp>#include <hpx/traits/is_launch_policy.hpp>#include <hpx/util/decay.hpp>#include <hpx/util/detail/pack.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <hpx/util/detail/pp/stringize.hpp>#include <hpx/parallel/executors/execution_parameters_fwd.hpp>#include <boost/ref.hpp>#include <cstddef>#include <functional>#include <type_traits>#include <utility>#include <vector>#include <hpx/parallel/executors/parallel_executor.hpp>#include <hpx/parallel/executors/rebind_executor.hpp>#include <hpx/parallel/executors/execution_fwd.hpp>#include <hpx/traits/executor_traits.hpp>#include <hpx/parallel/executors/sequenced_executor.hpp>#include <hpx/async_launch_policy_dispatch.hpp>#include <hpx/runtime/threads/thread_executor.hpp>#include <hpx/util/deferred_call.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/exception_list.hpp>#include <hpx/exception_list.hpp>#include <hpx/hpx_finalize.hpp>#include <hpx/util/assert.hpp>#include <exception>#include <iterator>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/traits/is_execution_policy.hpp>#include <memory>

file **auto_chunk_size.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/traits/is_executor_parameters.hpp>#include <hpx/util/high_resolution_clock.hpp>#include <hpx/util/steady_clock.hpp>#include <hpx/parallel/executors/execution_parameters.hpp>#include <algorithm>#include <cstddef>#include <cstdint>#include <type_traits>

file **dynamic_chunk_size.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/traits/is_executor_parameters.hpp>#include <cstddef>#include <type_traits>

file **execution_fwd.hpp**

#include <utility>#include <type_traits>#include <hpx/config.hpp>#include <hpx/traits/executor_traits.hpp>

file **execution_information_fwd.hpp**

#include <hpx/config.hpp>#include <hpx/parallel/executors/execution_fwd.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <hpx/traits/executor_traits.hpp>#include <cstddef>#include <type_traits>#include <utility>

file **guided_chunk_size.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include

<hpx/traits/is_executor_parameters.hpp>#include <algorithm>#include <cstddef>#include <type_traits>

file **parallel_executor.hpp**

#include <hpx/config.hpp>#include <hpx/async_launch_policy_dispatch.hpp>#include <hpx/lcos/future.hpp>#include <hpx/lcos/when_all_fwd.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include <hpx/parallel/executors/post_policy_dispatch.hpp>#include <hpx/parallel/executors/static_chunk_size.hpp>#include <hpx/runtime/get_worker_thread_num.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/runtime/threads/thread_helpers.hpp>#include <hpx/traits/future_traits.hpp>#include <hpx/traits/is_executor.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/bind.hpp>#include <hpx/util/range.hpp>#include <algorithm>#include <cstddef>#include <type_traits>#include <utility>#include <vector>

file **persistent_auto_chunk_size.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/traits/is_executor_parameters.hpp>#include <hpx/util/high_resolution_clock.hpp>#include <hpx/util/steady_clock.hpp>#include <algorithm>#include <cstddef>#include <cstdint>#include <type_traits>

file **sequenced_executor.hpp**

#include <hpx/config.hpp>#include <hpx/async_launch_policy_dispatch.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/threads/thread_executor.hpp>#include <hpx/traits/is_executor.hpp>#include <hpx/util/deferred_call.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/exception_list.hpp>#include <cstddef>#include <iterator>#include <type_traits>#include <utility>#include <vector>

file **service_executors.hpp**

#include <hpx/config.hpp>#include <hpx/lcos/future.hpp>#include <hpx/parallel/executors/static_chunk_size.hpp>#include <hpx/parallel/executors/thread_execution.hpp>#include <hpx/lcos/dataflow.hpp>#include <hpx/lcos/local/futures_factory.hpp>#include <hpx/runtime/threads/thread_executor.hpp>#include <hpx/traits/future_access.hpp>#include <hpx/traits/is_launch_policy.hpp>#include <hpx/util/bind.hpp>#include <hpx/util/bind_back.hpp>#include <hpx/util/detail/pack.hpp>#include <hpx/util/deferred_call.hpp>#include <hpx/util/range.hpp>#include <hpx/util/tuple.hpp>#include <hpx/util/unwrap.hpp>#include <hpx/parallel/executors/execution.hpp>#include <algorithm>#include <type_traits>#include <utility>#include <vector>#include <hpx/runtime/threads/executors/service_executors.hpp>#include <hpx/compat/condition_variable.hpp>#include <hpx/compat/mutex.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/threads/thread_enums.hpp>#include <hpx/throw_exception.hpp>#include <hpx/util/atomic_count.hpp>#include <hpx/util/steady_clock.hpp>#include <hpx/util/thread_description.hpp>#include <hpx/util/unique_function.hpp>#include <atomic>#include <chrono>#include <cstddef>#include <cstdint>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/traits/executor_traits.hpp>

file **static_chunk_size.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/traits/is_executor_parameters.hpp>#include <hpx/parallel/executors/execution_parameters_fwd.hpp>#include <cstddef>#include <type_traits>

file **thread_pool_executors.hpp**

#include <hpx/config.hpp>#include <hpx/lcos/future.hpp>#include <hpx/parallel/executors/execution_parameters.hpp>#include <hpx/parallel/executors/thread_execution.hpp>#include <hpx/parallel/executors/thread_execution_information.hpp>#include <hpx/runtime/get_os_thread_count.hpp>#include <hpx/runtime/threads/policies/scheduler_mode.hpp>#include <hpx/runtime/threads/thread_executor.hpp>#include <hpx/runtime/threads/topology.hpp>#include <hpx/traits/is_launch_policy.hpp>#include <hpx/parallel/executors/execution_information.hpp>#include <cstddef>#include <type_traits>#include <utility>#include <hpx/parallel/executors/thread_timed_execution.hpp>#include <hpx/lcos/local/packaged_task.hpp>#include <hpx/lcos/detail/future_data.hpp>#include <hpx/lcos/local/promise.hpp>#include <hpx/throw_exception.hpp>#include <hpx/traits/is_callable.hpp>#include <hpx/util/annotated_function.hpp>#include <hpx/util/thread_description.hpp>#include <hpx/util/unique_function.hpp>#include <exception>#include <memory>#include

<hpx/util/deferred_call.hpp>#include <hpx/util/steady_clock.hpp>#include <hpx/parallel/executors/timed_execution.hpp>#inclu
<hpx/parallel/executors/timed_execution_fwd.hpp>#include <hpx/parallel/executors/execution_fwd.hpp>#include <hpx/parallel/executors/timed_executors.hpp>#include <hpx/runtime/threads/thread.hpp>#include <hpx/traits/detail/wrap_int.hpp>#include <hpx/traits/executor_traits.hpp>#include <hpx/util/bind.hpp>#include <hpx/util/decay.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/executors/parallel_executor.hpp>#include <hpx/parallel/executors/sequenced_executor.hpp>#include <chrono>#include <functional>#include <hpx/traits/is_executor.hpp>#include <vector>#include <hpx/runtime/threads/executors/thread_pool_executors.hpp>#include <hpx/lcos/local/counting_semaphore.hpp>#include <hpx/lcos/local/detail/counting_semaphore.hpp>#include <hpx/lcos/local/detail/condition_variable.hpp>#include <hpx/lcos/local/spinlock.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/assert_owns_lock.hpp>#include <algorithm>#include <cstdint>#include <mutex>#include <hpx/runtime/resource/detail/partitioner.hpp>#include <hpx/runtime/resource/partitioner.hpp>#include <hpx/runtime/resource/partitioner_fwd.hpp>#include <hpx/runtime/resource/detail/create_partitioner.hpp>#include <hpx/runtime/runtime_mode.hpp>#include <hpx/util/bind_back.hpp>#include <hpx/util/find_prefix.hpp>#include <string>#include <hpx/util/function.hpp>#include <boost/program_options.hpp>#include <hpx/runtime/threads/cpu_mask.hpp>#include <hpx/runtime/threads/policies/affinity_data.hpp>#include <atomic>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/util/command_line_handling.hpp>#include <hpx/hpx_init.hpp>#include <hpx/hpx_finalize.hpp>#include <hpx/hpx_suspend.hpp>#include <hpx/runtime/shutdown_function.hpp>#include <hpx/runtime/startup_function.hpp>#include <boost/program_options/options_description.hpp>#include <boost/program_options/variables_map.hpp>#include <hpx/util/manage_config.hpp>#include <hpx/util/safe_lexical_cast.hpp>#include <boost/lexical_cast.hpp>#include <map>#include <hpx/util/runtime_configuration.hpp>#include <hpx/util/tuple.hpp>#include <iosfwd>#include <hpx/runtime/threads/thread_enums.hpp>

file **task_block.hpp**

#include <hpx/config.hpp>#include <hpx/async.hpp>#include <hpx/exception.hpp>#include <hpx/lcos/dataflow.hpp>#include <hpx/lcos/future.hpp>#include <hpx/lcos/local/spinlock.hpp>#include <hpx/lcos/when_all.hpp>#include <hpx/traits/is_future.hpp>#include <hpx/util/bind.hpp>#include <hpx/util/bind_back.hpp>#include <hpx/util/decay.hpp>#include <hpx/parallel/exception_list.hpp>#include <hpx/parallel/execution_policy.hpp>#include <hpx/parallel/executors/execution.hpp>#include <hpx/parallel/util/detail/algorithm_result.hpp>#include <boost/utility/addressof.hpp>#include <memory>#include <exception>#include <mutex>#include <type_traits>#include <utility>#include <vector>

file **manage_counter_type.hpp**

#include <hpx/config.hpp>#include <hpx/error_code.hpp>#include <hpx/performance_counters/counters_fwd.hpp>#include <hpx/util/function.hpp>#include <cstddef>#include <cstdint>#include <string>#include <vector>

file **basic_action.hpp**

#include <hpx/config.hpp>#include <hpx/exception.hpp>#include <hpx/lcos/sync_fwd.hpp>#include <hpx/runtime/actions/action_support.hpp>#include <hpx/runtime/actions/basic_action_fwd.hpp>#include <hpx/runtime/actions/continuation.hpp>#include <hpx/runtime/actions/detail/action_factory.hpp>#include <hpx/runtime/actions/detail/invocation_count_registry.hpp>#include <hpx/runtime/actions/preassigned_action_id.hpp>#include
<hpx/runtime/actions/transfer_action.hpp>#include <hpx/runtime/actions/transfer_continuation_action.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/naming/address.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/parcelset/detail/per_action_data_counter_registry.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <hpx/runtime/threads/thread_enums.hpp>#include <hpx/runtime_fwd.hpp>#include <hpx/traits/action_decorate_function.hpp>#include <hpx/traits/action_priority.hpp>#include <hpx/traits/action_remote_result.hpp>#include <hpx/traits/action_stacksize.hpp>#include <hpx/traits/is_action.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <hpx/traits/promise_local_result.hpp>#include <hpx/util/detail/pack.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <hpx/util/detail/pp/expand.hpp>#include <hpx/util/detail/pp/nargs.hpp>#include <hpx/util/detail/pp/stringize.hpp>#include <hpx/util/get_and_reset_value.hpp>#include <hpx/util/invoke_fused.hpp>#include

<hpx/util/logging.hpp>#include <hpx/util/tuple.hpp>#include <boost/utility/string_ref.hpp>#include <atomic>#include <cstddef>#include <cstdint>#include <exception>#include <sstream>#include <string>#include <type_traits>#include <utility>

Defines

HPX_REGISTER_ACTION_DECLARATION(...)

Declare the necessary component action boilerplate code.

The macro *HPX_REGISTER_ACTION_DECLARATION* can be used to declare all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to declare the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as ‘<’, ‘>’, or ‘:’.

```
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::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(...)

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.

file **component_action.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/actions/basic_action.hpp>#include <hpx/runtime/components/pinned_ptr.hpp>#include <hpx/runtime/naming/address.hpp>#include <hpx/traits/is_future.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <hpx/util/detail/pp/expand.hpp>#include <hpx/util/detail/pp/nargs.hpp>#include <boost/utility/string_ref.hpp>#include <cstdlib>#include <sstream>#include <string>#include <type_traits>#include <utility>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.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;
        }

        // 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.

file **plain_action.hpp**

#include <hpx/config.hpp>#include <hpx/runtime/actions/basic_action.hpp>#include <hpx/runtime/naming/address.hpp>#include <hpx/traits/component_type_database.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <hpx/util/detail/pp/expand.hpp>#include <hpx/util/detail/pp/nargs.hpp>#include <boost/utility/string_ref.hpp>#include <cstdlib>#include <sstream>#include <stdexcept>#include <string>#include <utility>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.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.

file **`applier_fwd.hpp`**

#include <hpx/config.hpp>

file **`basename_registration_fwd.hpp`**

#include <hpx/config.hpp>#include <hpx/components_fwd.hpp>#include <hpx/lcos_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/dataflow.hpp>#include <hpx/lcos/future.hpp>#include <hpx/performance_counters/performance_counter.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/util/bind_front.hpp>#include <hpx/performance_counters/counters_fwd.hpp>#include <hpx/performance_counters/stubs/performance_counter.hpp>#include <hpx/performance_counters/server/base_performance_counter.hpp>#include <hpx/lcos/base_lco_with_value.hpp>#include <hpx/performance_counters/counters.hpp>#include <hpx/performance_counters/performance_counter_base.hpp>#include <hpx/runtime/actions/component_action.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/runtime/components/server/component.hpp>#include <hpx/throw_exception.hpp>#include <hpx/util/atomic_count.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <string>#include <utility>#include <vector>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <hpx/runtime/serialization/string.hpp>#include <hpx/runtime/serialization/vector.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/bind_back.hpp>#include <hpx/util/unwrap.hpp>#include <algorithm>#include <cstddef>#include <cstdint>#include <iterator>#include <type_traits>

file **`colocating_distribution_policy.hpp`**

#include <hpx/config.hpp>#include <hpx/lcos/detail/async_colocated.hpp>#include <hpx/lcos/detail/async_colocated_callback.hpp>#include <hpx/lcos/detail/async_implementations.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/applier/detail/apply_colocated_callback_fwd.hpp>#include <hpx/runtime/applier/detail/apply_colocated_fwd.hpp>#include <hpx/runtime/applier/detail/apply_implementations.hpp>#inclu

<hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <hpx/traits/extract_action.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <hpx/traits/promise_local_result.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/lcos/async.hpp>#include <hpx/lcos/detail/async_colocated.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/actions/plain_action.hpp>#include <hpx/runtime/components/server/copy_component.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/traits/is_component.hpp>#include <type_traits>

file **default_distribution_policy.hpp**

#include <hpx/config.hpp>#include <hpx/lcos/dataflow.hpp>#include <hpx/lcos/future.hpp>#include <hpx/lcos/packaged_action.hpp>#include <hpx/runtime/actions/action_support.hpp>#include <hpx/runtime/applier/apply.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <hpx/runtime/serialization/vector.hpp>#include <hpx/runtime/serialization/shared_ptr.hpp>#include <hpx/traits/extract_action.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <hpx/traits/promise_local_result.hpp>#include <hpx/util/assert.hpp>#include <algorithm>#include <cstddef>#include <memory>#include <type_traits>#include <utility>#include <vector>

file **migrate_component.hpp**

#include <hpx/config.hpp>#include <hpx/lcos/async.hpp>#include <hpx/lcos/detail/async_colocated.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/actions/plain_action.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/server/migrate_component.hpp>#include <hpx/runtime/components/target_distribution_policy.hpp>#include <hpx/lcos/dataflow.hpp>#include <hpx/lcos/detail/async_implementations_fwd.hpp>#include <hpx/lcos/packaged_action.hpp>#include <hpx/runtime/actions/action_support.hpp>#include <hpx/runtime/agas/interface.hpp>#include

<hpx/runtime/applier/detail/apply_implementations_fwd.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <hpx/traits/extract_action.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <hpx/traits/promise_local_result.hpp>#include <algorithm>#include <cstddef>#include <type_traits>#include <utility>#include <vector>#include <hpx/traits/is_component.hpp>

file **new.hpp**
#include <hpx/config.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/default_distribution_policy.hpp>#include <hpx/runtime/components/server/create_component.hpp># <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/traits/is_client.hpp>#include <hpx/traits/is_component.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <hpx/util/lazy_enable_if.hpp>#include <algorithm>#include <cstddef>#include <type_traits>#include <utility>#include <vector>

file **find_here.hpp**
#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/naming/id_type.hpp>

file **find_localities.hpp**
#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <vector>

file **get_colocation_id.hpp**
#include <hpx/exception_fwd.hpp>#include <hpx/lcos_fwd.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/naming/id_type.hpp>

file **get_locality_id.hpp**
#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <cstdint>

file **get_locality_name.hpp**
#include <hpx/config.hpp>#include <hpx/lcos_fwd.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <string>

file **get_num_localities.hpp**
#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/lcos_fwd.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/components/component_type.hpp>#include <cstdint>

file **get_os_thread_count.hpp**
#include <hpx/config.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <cstddef>

file **get_ptr.hpp**
#include <hpx/config.hpp>#include <hpx/runtime_fwd.hpp>#include <hpx/runtime/agas/gva.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/runtime/get_lva.hpp>#include <hpx/runtime/launch_policy.hpp>#include <hpx/runtime/naming/address.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/throw_exception.hpp>#include <hpx/traits/component_pin_support.hpp>#include <hpx/traits/component_type_is_compatible.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/bind_back.hpp>#include <memory>

file **get_thread_name.hpp**
#include <hpx/config.hpp>#include <hpx/util/itt_notify.hpp>#include <string>

file **get_worker_thread_num.hpp**
#include <hpx/config.hpp>#include <hpx/error_code.hpp>#include <cstddef>

file **launch_policy.hpp**
#include <hpx/config.hpp>#include <hpx/runtime/threads/thread_enums.hpp>#include <hpx/runtime/serialization/serialization_fwd.hpp>#include <type_traits>#include <utility>

file `unmanaged.hpp`
#include <hpx/runtime/naming/name.hpp>

file `report_error.hpp`
#include <hpx/config.hpp>#include <cstddef>#include <exception>

file `partitioner.hpp`
#include <hpx/config.hpp>#include <hpx/runtime/resource/partitioner_fwd.hpp>#include <hpx/runtime/resource/detail/create_partitioner.hpp>#include <hpx/runtime/runtime_mode.hpp>#include <hpx/runtime/threads/policies/scheduler_mode.hpp>#include <hpx/util/function.hpp>#include <boost/program_options.hpp>#include <cstddef>#include <string>#include <utility>#include <vector>

file `partitioner_fwd.hpp`
#include <hpx/config.hpp>#include <hpx/runtime/threads/policies/callback_notifier.hpp>#include <hpx/runtime/threads_fwd.hpp>#include <hpx/util/function.hpp>#include <cstddef>#include <memory>#include <string>

file `runtime_mode.hpp`
#include <hpx/config.hpp>#include <string>

file `set_parcel_write_handler.hpp`
#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/parcelset_fwd.hpp>#include <hpx/util/function.hpp>#include <boost/system/error_code.hpp>

file `shutdown_function.hpp`
#include <hpx/config.hpp>#include <hpx/util/unique_function.hpp>

file `startup_function.hpp`
#include <hpx/config.hpp>#include <hpx/util/unique_function.hpp>

file `scheduler_mode.hpp`

file `thread_data_fwd.hpp`
#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/threads/coroutines/coroutine_fwd.hpp>#in <hpx/runtime/threads/thread_enums.hpp>#include <hpx/runtime/threads/thread_id_type.hpp>#include <hpx/util_fwd.hpp>#include <hpx/util/function.hpp>#include <hpx/util/unique_function.hpp>#include <cstddef>#include <cstdint>#include <utility>#include <memory>

file `thread_enums.hpp`
#include <hpx/config.hpp>#include <hpx/runtime/threads/detail/combined_tagged_state.hpp>#include <cstddef>#include <cstdint>

file `thread_helpers.hpp`
#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/naming_fwd.hpp>#include <hpx/runtime/threads_fwd.hpp>#include <hpx/runtime/thread_pool_helpers.hpp>#include <hpx/runtime/threads/policies/scheduler_mode.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <hpx/runtime/threads/thread_enums.hpp>#include <hpx/util_fwd.hpp>#include <hpx/util/unique_function.hpp>#include <hpx/util/register_locks.hpp>#include <hpx/util/steady_clock.hpp>#include <hpx/util/thread_description.hpp>#include <atomic>#include <chrono>#include <cstddef>#include <cstdint>#include <type_traits>#include <utility>

file `thread_pool_base.hpp`
#include <hpx/config.hpp>#include <hpx/compat/barrier.hpp>#include <hpx/compat/condition_variable.hpp>#include <hpx/compat/mutex.hpp>#include <climits>#include <cstddef>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/compat/thread.hpp>#include <hpx/error_code.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/lcos/future.hpp>#include <hpx/lcos/local/no_mutex.hpp>#include <hpx/lcos/local/spinlock.hpp>#include <hpx/runtime/thread_pool_helpers.hpp>#include <hpx/runtime/threads/cpu_mask.hpp>#include <hpx/runtime/threads/policies/affinity_data.hpp>#include <hpx/runtime/threads/policies/callback_notifier.hpp>#include <hpx/runtime/threads/policies/scheduler_mode.hpp>#include <hpx/runtime/threads/thread_executor.hpp>#include <hpx/runtime/threads/thread_init_data.hpp>#include

<hpx/runtime/threads/topology.hpp>#include <hpx/state.hpp>#include <hpx/util/steady_clock.hpp>#include <hpx/util_fwd.hpp>#include <cstdint>#include <exception>#include <functional>#include <iosfwd>#include <memory>#include <mutex>#include <string>#include <vector>

file **trigger_lco.hpp**

#include <hpx/config.hpp>#include <hpx/lcos_fwd.hpp>#include <hpx/runtime/actions/continuation_fwd.hpp>#include <hpx/runtime/actions/action_priority.hpp>#include <hpx/runtime/applier/detail/apply_implementations_fwd.hpp>#include <hpx/runtime/naming/address.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/util/assert.hpp>#include <hpx/util/decay.hpp>#include <exception>#include <type_traits>#include <utility>

file **runtime_fwd.hpp**

#include <hpx/config.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/runtime/basename_registration_fwd.hpp>#include <hpx/runtime/config_entry.hpp>#include <hpx/runtime/find_localities.hpp>#include <hpx/runtime/get_colocation_id.hpp>#include <hpx/runtime/get_locality_id.hpp>#include <hpx/runtime/get_locality_name.hpp>#include <hpx/runtime/get_num_localities.hpp>#include <hpx/runtime/get_os_thread_count.hpp>#include <hpx/runtime/get_thread_name.hpp>#include <hpx/runtime/get_worker_thread_num.hpp>#include <hpx/runtime/naming_fwd.hpp>#include <hpx/runtime/report_error.hpp>#include <hpx/runtime/runtime_fwd.hpp>#include <hpx/runtime/runtime_mode.hpp>#include <hpx/runtime/set_parcel_write_handler.hpp>#include <hpx/runtime/shutdown_function.hpp>#include <hpx/runtime/startup_function.hpp>#include <hpx/util/function.hpp>#include <hpx/util_fwd.hpp>#include <cstddef>#include <cstdint>#include <string>

file **throw_exception.hpp**

#include <hpx/config.hpp>#include <hpx/error.hpp>#include <hpx/exception_fwd.hpp>#include <hpx/util/detail/pp/cat.hpp>#include <hpx/util/detail/pp/expand.hpp>#include <hpx/util/detail/pp/nargs.hpp>#include <boost/current_function.hpp>#include <boost/system/error_code.hpp>#include <exception>#include <string>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.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.

```
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.

file **is_execution_policy.hpp**

#include <hpx/config.hpp>#include <hpx/util/decay.hpp>#include <type_traits>

file **checkpoint.hpp**

#include <hpx/dataflow.hpp>#include <hpx/lcos/future.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include <hpx/runtime/serialization/vector.hpp>#include <cstddef>#include <fstream>#include <iosfwd>#include <sstream>#include <string>#include <type_traits>#include <utility>#include <vector> 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.

file **debugging.hpp**

#include <hpx/config.hpp>

file **invoke.hpp**

#include <hpx/config.hpp>#include <hpx/util/result_of.hpp>#include <hpx/util/void_guard.hpp>#include <boost/ref.hpp>#include <functional>#include <type_traits>#include <utility>

Defines

HPX_INVOKE (F, ...)

HPX_INVOKE_R (R, F, ...)

file **invoke_fused.hpp**

#include <hpx/config.hpp>#include <hpx/util/detail/pack.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/result_of.hpp>#include <hpx/util/tuple.hpp>#include <hpx/util/void_guard.hpp>#include <cstddef>#include <type_traits>#include <utility>

file **pack_traversal.hpp**

#include <hpx/util/detail/pack_traversal_impl.hpp>#include <hpx/util/tuple.hpp>#include <type_traits>#include <utility>

file **pack_traversal_async.hpp**

#include <hpx/util/detail/pack_traversal_async_impl.hpp>#include <utility>

file **unwrap.hpp**

#include <hpx/config.hpp>#include <hpx/util/detail/unwrap_impl.hpp>#include <cstddef>#include <utility>

file **unwrapped.hpp**

#include <hpx/config.hpp>

dir **/hpx/source/hpx/runtime/actions**

dir **/hpx/source/hpx/parallel/algorithms**

dir **/hpx/source/hpx/components/component_storage**

dir **/hpx/source/hpx/components**

dir **/hpx/source/hpx/runtime/components**

dir **/hpx/source/hpx/parallel/container_algorithms**

dir **/hpx/source/hpx/parallel/executors**

dir **/hpx/source/hpx**

dir **/hpx/source/hpx/lcos**

dir **/hpx/source/hpx/runtime/naming**

dir **/hpx/source/hpx/parallel**

dir **/hpx/source/hpx/performance_counters**

dir **/hpx/source/hpx/runtime/threads/policies**

dir **/hpx/source/hpx/runtime/resource**

dir **/hpx/source/hpx/runtime**

dir **/hpx/source**

dir **/hpx/source/hpx/runtime/threads**

dir **/hpx/source/hpx/traits**

dir **/hpx/source/hpx/util**

2.9 Contributing to *HPX*

HPX development happens on Github. The following sections are a collection of useful information related to *HPX* development.

2.9.1 Release procedure for *HPX*

Below is a step-wise 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 practice 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. Make a list of examples and benchmarks that should not go into the release. Build all examples and benchmarks that will go in the release and make sure they build and run as expected.
   - Make sure all examples and benchmarks have example input files, and usage documentation, either in the form of comments or a readme.
3. Send the list of examples and benchmarks that will be included in the release to hpx-users@stellar.cct.lsu.edu and stellar@cct.lsu.edu, and ask for feedback. Update the list as necessary.
4. 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. Add the new release notes to the table of contents in `docs/sphinx/releases.rst`.
5. 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

6. Change the APEX release branch to be the most current release tag available in the `git_external` APEX section of the main `CMakeLists.txt`. Please contact the maintainers at the APEX repository[255] to generate a new release to synchronize with the HPX release.
7. Change the hpxMP release branch to be the most current release tag available in the `git_external` hpxMP section of the main `CMakeLists.txt`. Please contact the maintainers at the hpxMP repository[256] to generate a new release to synchronize with the HPX release.
8. 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.
9. Unprotect the release branch in the github repository settings so that it can be deleted and recreated.
10. Delete the old release branch, and create a new one by branching a stable point from master. If you are creating a patch release, branch from the release tag for which you want to create a patch release.
    - `git push origin --delete release`
    - `git branch -D release`
    - `git checkout [stable point in master]`
    - `git branch release`
    - `git push origin release`
    - `git branch --set-upstream-to=origin/release release`
11. Protect the release branch again to disable deleting and force pushes.
12. Checkout the release branch, and replace the `-trunk` tag in `CMakeLists.txt` with `-rc1`.
13. Remove the examples and benchmarks that will not go into the release from the release branch.
14. 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. This step does not apply to patch releases.

    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 which allows to explicitly disable 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.
15. Tag a release candidate from the release branch, where tag name is the version to be released with a "-rcN" suffix and description is "HPX V$VERSION: The C++ Standards Library for Parallelism and Concurrency".

[255] http://github.com/khuck/xpress-apex
[256] https://github.com/STEllAR-GROUP/hpxMP

- `git tag -a [tag name] -m '[description]'`
- `git push origin [tag name]`
- Create a pre-release on GitHub

16. Switch Buildbot over to test the release branch
    - `https://github.com/STEllAR-GROUP/hermione-buildbot/blob/rostam/master/master.cfg`
    - Line 120
17. 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.
18. 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.
    - Repeat by tagging a new release candidate as many times as needed.
19. Checkout the release branch, and replace the `-rcN` tag in `CMakeLists.txt` with an empty string.
20. 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.
21. 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.
22. 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`.
23. Tag the release from the release branch, where tag name is the version to be released and description is "HPX V$VERSION: The C++ Standards Library for Parallelism and Concurrency". Sign the release tag with the `contact@stellar-group.org` key by adding the `-s` flag to `git tag`. Make sure you change git to sign with the `contact@stellar-group.org` key, rather than your own key if you use one. You also need to change the name and email used for commits. Change them to `STE||AR Group` and `contact@stellar-group.org`, respectively. Finally, the `contact@stellar-group.org` email address needs to be added to your GitHub account for the tag to show up as verified.
    - `git tag -s -a [tag name] -m '[description]'`
    - `git push origin [tag name]`
24. Create a release on github
    - Refer to 'What's New' section in the documentation you uploaded in the notes for the Github release (see previous releases for a hint).
    - A DOI number using Zenodo is automatically assigned once the release is created as such on github.
    - Verify on Zenodo (https://zenodo.org/) that release was uploaded. Logging into zenodo using the github credentials might be necessary to see the new release as it usually takes a while for it to propagate to the search engine used on zenodo.
25. Roll a release candidate using `tools/roll_release.sh` (from root directory), and add the hashsums generated by the script to the "downloads" page of the website.
26. Upload the packages the website. Use the following formats:

```
http://stellar.cct.lsu.edu/files/hpx_#.#.#.zip
http://stellar.cct.lsu.edu/files/hpx_#.#.#.tar.gz
```

```
http://stellar.cct.lsu.edu/files/hpx_#.#.#.tar.bz2
http://stellar.cct.lsu.edu/files/hpx_#.#.#.7z
```

27. Update the websites (stellar-group.org[257] and stellar.cct.lsu.edu[258]) with the following:
    - Download links on the download page
    - Documentation links on the docs page (link to generated documentation on GitHub Pages)
    - 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)
28. Merge release branch into master.
29. Create a new branch from master, and check that branch out (name it for example by the next version number). Bump the HPX version to the next release target. The following files contain version info:
    - `CMakeLists.txt`
    - Grep for old version number
30. Create a new "What's New" section for the docs of the next anticipated release. Set the date to "unreleased".
31. Merge new branch containing next version numbers to master, resolve conflicts if necessary.
32. Switch Buildbot back to test the main branch
    - `https://github.com/STEllAR-GROUP/hermione-buildbot/blob/rostam/master/master.cfg`
    - Line 120
33. 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
34. 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.
35. Beer and pizza.

2.9.2 Testing *HPX*

To ensure correctness of *HPX* we ship a large variety of unit and regression tests. The tests are driven by the CTest[259] tool and are executed automatically by buildbot (see *HPX* Buildbot Website[260]) on each commit to the *HPX* Github[261] 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[262] tracker with detailed information about the target system.

[257] https://stellar-group.org
[258] https://stellar.cct.lsu.edu
[259] https://gitlab.kitware.com/cmake/community/wikis/doc/ctest/Testing-With-CTest
[260] https://rostam.cct.lsu.edu/
[261] https://github.com/STEllAR-GROUP/hpx/
[262] 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[263] for further details.

Issue tracker

If you stumble over a bug or missing feature missing feature in *HPX* please submit an issue to our *HPX* Issues[264]. For more information on how to submit support requests or other means of getting in contact with the developers please see the Support Website[265].

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[266]. We also run tests on all pull requests using both CircleCI[267] and a combination of CDash[268] and pycicle[269]. You can see the dashboards here: CircleCI HPX dashboard[270] and CDash HPX dashboard[271] .

2.9.3 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[272] provides a convenient alternative to quickly get all the dependencies needed to start development of *HPX*. Our testing setup on CircleCI[273] uses a docker image to run all tests.

To get started you need to install Docker[274] using whatever means is most convenient on your system. Once you have Docker[275] 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[276]. 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[277] for more information on using Docker[278].

[263] https://www.cmake.org/cmake/help/latest/manual/ctest.1.html
[264] https://github.com/STEllAR-GROUP/hpx/issues
[265] https://stellar.cct.lsu.edu/support/
[266] https://rostam.cct.lsu.edu/
[267] https://circleci.com
[268] https://www.kitware.com/cdash/project/about.html
[269] https://github.com/biddisco/pycicle/
[270] https://circleci.com/gh/STEllAR-GROUP/hpx
[271] https://cdash.cscs.ch/index.php?project=HPX
[272] https://www.docker.com
[273] https://circleci.com
[274] https://www.docker.com
[275] https://www.docker.com
[276] https://hub.docker.com/r/stellargroup/build_env/
[277] https://docs.docker.com/
[278] 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)[279]).

2.9.4 Documentation

This documentation is built using Sphinx[280], and an automatically generated API reference using Doxygen[281] and Breathe[282].

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[283] configuration. To build the documentation build the `docs` target using your build tool.

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.

[279] https://docs.docker.com/storage/bind-mounts/
[280] http://www.sphinx-doc.org
[281] https://www.doxygen.org
[282] https://breathe.readthedocs.io/en/latest
[283] https://www.cmake.org

API documentation

The source code is documented using Doxygen[284]. 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 Releases

2.10.1 *HPX* V1.3.0 (unreleased)

General changes

Breaking changes

- Executable and library targets are now created without the `_exe` and `_lib` suffix respectively. For example, the target `hello_world_exe` is now simply called `hello_world`.

Closed issues

Closed pull requests

2.10.2 *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[285] - Build HPX 1.2 with boost 1.69
- Issue #3635[286] - Non-deterministic crashing on Stampede2
- Issue #3550[287] - 1>e:000workhpxsrcthrow_exception.cpp(54): error C2440: '<function-style-cast>': cannot convert from 'boost::system::error_code' to 'hpx::exception'
- Issue #3549[288] - HPX 1.2.0 does not build on i686, but release candidate did
- Issue #3511[289] - Build on s390x fails

[284] https://www.doxygen.org
[285] https://github.com/STEllAR-GROUP/hpx/issues/3638
[286] https://github.com/STEllAR-GROUP/hpx/issues/3635
[287] https://github.com/STEllAR-GROUP/hpx/issues/3550
[288] https://github.com/STEllAR-GROUP/hpx/issues/3549
[289] https://github.com/STEllAR-GROUP/hpx/issues/3511

- Issue #3509[290] - Build on armv7l fails

Closed pull requests

- PR #3695[291] - Don't install CMake templates and packaging files
- PR #3666[292] - Fixing yet another race in future_data
- PR #3663[293] - Fixing race between setting and getting the value inside future_data
- PR #3648[294] - Adding timestamp option for S390x platform
- PR #3647[295] - Blind attempt to fix warnings issued by gcc V9
- PR #3611[296] - Include GNUInstallDirs earlier to have it available for subdirectories
- PR #3595[297] - Use GNUInstallDirs lib path in pkgconfig config file
- PR #3593[298] - Add include(GNUInstallDirs) to HPXMacros.cmake
- PR #3591[299] - Fix compilation error on arm7 architecture. Compiles and runs on Fedora 29 on Pi 3.
- PR #3558[300] - Adding constructor *exception(boost::system::error_code const&)*
- PR #3555[301] - cmake: make install locations configurable
- PR #3551[302] - Fix uint64_t causing compilation fail on i686

2.10.3 *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[303] or pkg-config. The old behaviour can be restored by setting `HPX_WITH_DYNAMIC_HPX_MAIN=OFF` during CMake[304] 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.

290 https://github.com/STEllAR-GROUP/hpx/issues/3509
291 https://github.com/STEllAR-GROUP/hpx/pull/3695
292 https://github.com/STEllAR-GROUP/hpx/pull/3666
293 https://github.com/STEllAR-GROUP/hpx/pull/3663
294 https://github.com/STEllAR-GROUP/hpx/pull/3648
295 https://github.com/STEllAR-GROUP/hpx/pull/3647
296 https://github.com/STEllAR-GROUP/hpx/pull/3611
297 https://github.com/STEllAR-GROUP/hpx/pull/3595
298 https://github.com/STEllAR-GROUP/hpx/pull/3593
299 https://github.com/STEllAR-GROUP/hpx/pull/3591
300 https://github.com/STEllAR-GROUP/hpx/pull/3558
301 https://github.com/STEllAR-GROUP/hpx/pull/3555
302 https://github.com/STEllAR-GROUP/hpx/pull/3551
303 https://www.cmake.org
304 https://www.cmake.org

- We have significantly improved the performance of our futures implementation by making the shared state atomic.
- We have replaced Boostbook by Sphinx for our documentation. This means the documentation is easier to navigate with built-in search and table of contents. We have also added a quick start section and restructured the documentation to be easier to follow for new users.
- We have added a new option to the `--hpx:threads` command line option. It is now possible to use `cores` to tell *HPX* to only use one worker thread per core, unlike the existing option `all` which uses one worker thread per processing unit (processing unit can be a hyperthread if hyperthreads are available). The default value of `--hpx:threads` has also been changed to `cores` as this leads to better performance in most cases.
- All command line options can now be passed alongside configuration options when initializing *HPX*. This means that some options that were previously only available on the command line can now be set as configuration options.
- HPXMP is a portable, scalable, and flexible application programming interface using the OpenMP specification that supports multi-platform shared memory multiprocessing programming in C and C++. HPXMP can be enabled within *HPX* by setting `DHPX_WITH_HPXMP=ON` during CMake[305] 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[306] 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[307] 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[308].
- 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`.

305 https://www.cmake.org
306 https://www.cmake.org
307 https://www.cmake.org
308 https://github.com/STEllAR-GROUP/hpx/pull/3266

Closed issues

- Issue #3538[309] - numa handling incorrect for hwloc 2
- Issue #3533[310] - Cmake version 3.5.1does not work (git ff26b35 2018-11-06)
- Issue #3526[311] - Failed building hpx-1.2.0-rc1 on Ubuntu16.04 x86-64 Virtualbox VM
- Issue #3512[312] - Build on aarch64 fails
- Issue #3475[313] - HPX fails to link if the MPI parcelport is enabled
- Issue #3462[314] - CMake configuration shows a minor and inconsequential failure to create a symlink
- Issue #3461[315] - Compilation Problems with the most recent Clang
- Issue #3460[316] - Deadlock when create_partitioner fails (assertion fails) in debug mode
- Issue #3455[317] - HPX build failing with HWLOC errors on POWER8 with hwloc 1.8
- Issue #3438[318] - HPX no longer builds on IBM POWER8
- Issue #3426[319] - hpx build failed on MacOS
- Issue #3424[320] - CircleCI builds broken for forked repositories
- Issue #3422[321] - Benchmarks in tests.performance.local are not run nightly
- Issue #3408[322] - CMake Targets for HPX
- Issue #3399[323] - processing unit out of bounds
- Issue #3395[324] - Floating point bug in hpx/runtime/threads/policies/scheduler_base.hpp
- Issue #3378[325] - compile error with lcos::communicator
- Issue #3376[326] - Failed to build HPX with APEX using clang
- Issue #3366[327] - Adapted Safe_Object example fails for –hpx:threads > 1
- Issue #3360[328] - Segmentation fault when passing component id as parameter
- Issue #3358[329] - HPX runtime hangs after multiple (~thousands) start-stop sequences
- Issue #3352[330] - Support TCP provider in libfabric ParcelPort

309 https://github.com/STEllAR-GROUP/hpx/issues/3538
310 https://github.com/STEllAR-GROUP/hpx/issues/3533
311 https://github.com/STEllAR-GROUP/hpx/issues/3526
312 https://github.com/STEllAR-GROUP/hpx/issues/3512
313 https://github.com/STEllAR-GROUP/hpx/issues/3475
314 https://github.com/STEllAR-GROUP/hpx/issues/3462
315 https://github.com/STEllAR-GROUP/hpx/issues/3461
316 https://github.com/STEllAR-GROUP/hpx/issues/3460
317 https://github.com/STEllAR-GROUP/hpx/issues/3455
318 https://github.com/STEllAR-GROUP/hpx/issues/3438
319 https://github.com/STEllAR-GROUP/hpx/issues/3426
320 https://github.com/STEllAR-GROUP/hpx/issues/3424
321 https://github.com/STEllAR-GROUP/hpx/issues/3422
322 https://github.com/STEllAR-GROUP/hpx/issues/3408
323 https://github.com/STEllAR-GROUP/hpx/issues/3399
324 https://github.com/STEllAR-GROUP/hpx/issues/3395
325 https://github.com/STEllAR-GROUP/hpx/issues/3378
326 https://github.com/STEllAR-GROUP/hpx/issues/3376
327 https://github.com/STEllAR-GROUP/hpx/issues/3366
328 https://github.com/STEllAR-GROUP/hpx/issues/3360
329 https://github.com/STEllAR-GROUP/hpx/issues/3358
330 https://github.com/STEllAR-GROUP/hpx/issues/3352

- Issue #3342[331] - undefined reference to __atomic_load_16
- Issue #3339[332] - setting command line options/flags from init cfg is not obvious
- Issue #3325[333] - AGAS migrates components prematurely
- Issue #3321[334] - hpx bad_parameter handling is awful
- Issue #3318[335] - Benchmarks fail to build with C++11
- Issue #3304[336] - hpx::threads::run_as_hpx_thread does not properly handle exceptions
- Issue #3300[337] - Setting pu step or offset results in no threads in default pool
- Issue #3297[338] - Crash with APEX when running Phylanx lra_csv with > 1 thread
- Issue #3296[339] - Building HPX with APEX configuration gives compiler warnings
- Issue #3290[340] - make tests failing at hello_world_component
- Issue #3285[341] - possible compilation error when "using namespace std;" is defined before including "hpx" headers files
- Issue #3280[342] - HPX fails on OSX
- Issue #3272[343] - CircleCI does not upload generated docker image any more
- Issue #3270[344] - Error when compiling CUDA examples
- Issue #3267[345] - `tests.unit.host_.block_allocator` fails occasionally
- Issue #3264[346] - Possible move to Sphinx for documentation
- Issue #3263[347] - Documentation improvements
- Issue #3259[348] - `set_parcel_write_handler` test fails occasionally
- Issue #3258[349] - Links to source code in documentation are broken
- Issue #3247[350] - Rare `tests.unit.host_.block_allocator` test failure on 1.1.0-rc1
- Issue #3244[351] - Slowing down and speeding up an interval_timer
- Issue #3215[352] - Cannot build both tests and examples on MSVC with pseudo-dependencies enabled
- Issue #3195[353] - Unnecessary customization point route causing performance penalty

[331] https://github.com/STEllAR-GROUP/hpx/issues/3342
[332] https://github.com/STEllAR-GROUP/hpx/issues/3339
[333] https://github.com/STEllAR-GROUP/hpx/issues/3325
[334] https://github.com/STEllAR-GROUP/hpx/issues/3321
[335] https://github.com/STEllAR-GROUP/hpx/issues/3318
[336] https://github.com/STEllAR-GROUP/hpx/issues/3304
[337] https://github.com/STEllAR-GROUP/hpx/issues/3300
[338] https://github.com/STEllAR-GROUP/hpx/issues/3297
[339] https://github.com/STEllAR-GROUP/hpx/issues/3296
[340] https://github.com/STEllAR-GROUP/hpx/issues/3290
[341] https://github.com/STEllAR-GROUP/hpx/issues/3285
[342] https://github.com/STEllAR-GROUP/hpx/issues/3280
[343] https://github.com/STEllAR-GROUP/hpx/issues/3272
[344] https://github.com/STEllAR-GROUP/hpx/issues/3270
[345] https://github.com/STEllAR-GROUP/hpx/issues/3267
[346] https://github.com/STEllAR-GROUP/hpx/issues/3264
[347] https://github.com/STEllAR-GROUP/hpx/issues/3263
[348] https://github.com/STEllAR-GROUP/hpx/issues/3259
[349] https://github.com/STEllAR-GROUP/hpx/issues/3258
[350] https://github.com/STEllAR-GROUP/hpx/issues/3247
[351] https://github.com/STEllAR-GROUP/hpx/issues/3244
[352] https://github.com/STEllAR-GROUP/hpx/issues/3215
[353] https://github.com/STEllAR-GROUP/hpx/issues/3195

- Issue #3088[354] - A strange thing in parallel::sort.
- Issue #2650[355] - libfabric support for passive endpoints
- Issue #1205[356] - TSS is broken

Closed pull requests

- PR #3542[357] - Fix numa lookup from pu when using hwloc 2.x
- PR #3541[358] - Fixing the build system of the MPI parcelport
- PR #3540[359] - Updating HPX people section
- PR #3539[360] - Splitting test to avoid OOM on CircleCI
- PR #3537[361] - Fix guided exec
- PR #3536[362] - Updating grants which support the LSU team
- PR #3535[363] - Fix hiding of docker credentials
- PR #3534[364] - Fixing #3533
- PR #3532[365] - fixing minor doc typo –hpx:print-counter-at arg
- PR #3530[366] - Changing APEX default tag to v2.1.0
- PR #3529[367] - Remove leftover security options and documentation
- PR #3528[368] - Fix hwloc version check
- PR #3524[369] - Do not build guided pool examples with older GCC compilers
- PR #3523[370] - Fix logging regression
- PR #3522[371] - Fix more warnings
- PR #3521[372] - Fixing argument handling in induction and reduction clauses for parallel::for_loop
- PR #3520[373] - Remove docs symlink and versioned docs folders
- PR #3519[374] - hpxMP release
- PR #3518[375] - Change all steps to use new docker image on CircleCI

354 https://github.com/STEllAR-GROUP/hpx/issues/3088
355 https://github.com/STEllAR-GROUP/hpx/issues/2650
356 https://github.com/STEllAR-GROUP/hpx/issues/1205
357 https://github.com/STEllAR-GROUP/hpx/pull/3542
358 https://github.com/STEllAR-GROUP/hpx/pull/3541
359 https://github.com/STEllAR-GROUP/hpx/pull/3540
360 https://github.com/STEllAR-GROUP/hpx/pull/3539
361 https://github.com/STEllAR-GROUP/hpx/pull/3537
362 https://github.com/STEllAR-GROUP/hpx/pull/3536
363 https://github.com/STEllAR-GROUP/hpx/pull/3535
364 https://github.com/STEllAR-GROUP/hpx/pull/3534
365 https://github.com/STEllAR-GROUP/hpx/pull/3532
366 https://github.com/STEllAR-GROUP/hpx/pull/3530
367 https://github.com/STEllAR-GROUP/hpx/pull/3529
368 https://github.com/STEllAR-GROUP/hpx/pull/3528
369 https://github.com/STEllAR-GROUP/hpx/pull/3524
370 https://github.com/STEllAR-GROUP/hpx/pull/3523
371 https://github.com/STEllAR-GROUP/hpx/pull/3522
372 https://github.com/STEllAR-GROUP/hpx/pull/3521
373 https://github.com/STEllAR-GROUP/hpx/pull/3520
374 https://github.com/STEllAR-GROUP/hpx/pull/3519
375 https://github.com/STEllAR-GROUP/hpx/pull/3518

- PR #3516[376] - Drop usage of deprecated facilities removed in C++17
- PR #3515[377] - Remove remaining uses of Boost.TypeTraits
- PR #3513[378] - Fixing a CMake problem when trying to use libfabric
- PR #3508[379] - Remove memory_block component
- PR #3507[380] - Propagating the MPI compile definitions to all relevant targets
- PR #3503[381] - Update documentation colors and logo
- PR #3502[382] - Fix bogus ‘throws‘ bindings in scheduled_thread_pool_impl
- PR #3501[383] - Split parallel::remove_if tests to avoid OOM on CircleCI
- PR #3500[384] - Support NONAMEPREFIX in add_hpx_library()
- PR #3497[385] - Note that cuda support requires cmake 3.9
- PR #3495[386] - Fixing dataflow
- PR #3493[387] - Remove deprecated options for 1.2.0 part 2
- PR #3492[388] - Add CUDA_LINK_LIBRARIES_KEYWORD to allow PRIVATE keyword in linkage t…
- PR #3491[389] - Changing Base docker image
- PR #3490[390] - Don’t create tasks immediately with hpx::apply
- PR #3489[391] - Remove deprecated options for 1.2.0
- PR #3488[392] - Revert “Use BUILD_INTERFACE generator expression to fix cmake flag exports”
- PR #3487[393] - Revert “Fixing type attribute warning for transfer_action”
- PR #3485[394] - Use BUILD_INTERFACE generator expression to fix cmake flag exports
- PR #3483[395] - Fixing type attribute warning for transfer_action
- PR #3481[396] - Remove unused variables
- PR #3480[397] - Towards a more lightweigh transfer action
- PR #3479[398] - Fix FLAGS - Use correct version of target_compile_options

[376] https://github.com/STEllAR-GROUP/hpx/pull/3516
[377] https://github.com/STEllAR-GROUP/hpx/pull/3515
[378] https://github.com/STEllAR-GROUP/hpx/pull/3513
[379] https://github.com/STEllAR-GROUP/hpx/pull/3508
[380] https://github.com/STEllAR-GROUP/hpx/pull/3507
[381] https://github.com/STEllAR-GROUP/hpx/pull/3503
[382] https://github.com/STEllAR-GROUP/hpx/pull/3502
[383] https://github.com/STEllAR-GROUP/hpx/pull/3501
[384] https://github.com/STEllAR-GROUP/hpx/pull/3500
[385] https://github.com/STEllAR-GROUP/hpx/pull/3497
[386] https://github.com/STEllAR-GROUP/hpx/pull/3495
[387] https://github.com/STEllAR-GROUP/hpx/pull/3493
[388] https://github.com/STEllAR-GROUP/hpx/pull/3492
[389] https://github.com/STEllAR-GROUP/hpx/pull/3491
[390] https://github.com/STEllAR-GROUP/hpx/pull/3490
[391] https://github.com/STEllAR-GROUP/hpx/pull/3489
[392] https://github.com/STEllAR-GROUP/hpx/pull/3488
[393] https://github.com/STEllAR-GROUP/hpx/pull/3487
[394] https://github.com/STEllAR-GROUP/hpx/pull/3485
[395] https://github.com/STEllAR-GROUP/hpx/pull/3483
[396] https://github.com/STEllAR-GROUP/hpx/pull/3481
[397] https://github.com/STEllAR-GROUP/hpx/pull/3480
[398] https://github.com/STEllAR-GROUP/hpx/pull/3479

- PR #3478[399] - Making sure the application's exit code is properly propagated back to the OS
- PR #3476[400] - Don't print docker credentials as part of the environment.
- PR #3473[401] - Fixing invalid cmake code if no jemalloc prefix was given
- PR #3472[402] - Attempting to work around recent clang test compilation failures
- PR #3471[403] - Enable jemalloc on windows
- PR #3470[404] - Updates readme
- PR #3468[405] - Avoid hang if there is an exception thrown during startup
- PR #3467[406] - Add compiler specific fallthrough attributes if C++17 attribute is not available
- PR #3466[407] - - bugfix : fix compilation with llvm-7.0
- PR #3465[408] - This patch adds various optimizations extracted from the thread_local_allocator work
- PR #3464[409] - Check for forked repos in CircleCI docker push step
- PR #3463[410] - - cmake : create the parent directory before symlinking
- PR #3459[411] - Remove unused/incomplete functionality from util/logging
- PR #3458[412] - Fix a problem with scope of CMAKE_CXX_FLAGS and hpx_add_compile_flag
- PR #3457[413] - Fixing more size_t -> int16_t (and similar) warnings
- PR #3456[414] - Add #ifdefs to topology.cpp to support old hwloc versions again
- PR #3454[415] - Fixing warnings related to silent conversion of size_t –> int16_t
- PR #3451[416] - Add examples as unit tests
- PR #3450[417] - Constexpr-fying bind and other functional facilities
- PR #3446[418] - Fix some thread suspension timeouts
- PR #3445[419] - Fix various warnings
- PR #3443[420] - Only enable service pool config options if pools are enabled
- PR #3441[421] - Fix missing closing brackets in documentation

[399] https://github.com/STEllAR-GROUP/hpx/pull/3478
[400] https://github.com/STEllAR-GROUP/hpx/pull/3476
[401] https://github.com/STEllAR-GROUP/hpx/pull/3473
[402] https://github.com/STEllAR-GROUP/hpx/pull/3472
[403] https://github.com/STEllAR-GROUP/hpx/pull/3471
[404] https://github.com/STEllAR-GROUP/hpx/pull/3470
[405] https://github.com/STEllAR-GROUP/hpx/pull/3468
[406] https://github.com/STEllAR-GROUP/hpx/pull/3467
[407] https://github.com/STEllAR-GROUP/hpx/pull/3466
[408] https://github.com/STEllAR-GROUP/hpx/pull/3465
[409] https://github.com/STEllAR-GROUP/hpx/pull/3464
[410] https://github.com/STEllAR-GROUP/hpx/pull/3463
[411] https://github.com/STEllAR-GROUP/hpx/pull/3459
[412] https://github.com/STEllAR-GROUP/hpx/pull/3458
[413] https://github.com/STEllAR-GROUP/hpx/pull/3457
[414] https://github.com/STEllAR-GROUP/hpx/pull/3456
[415] https://github.com/STEllAR-GROUP/hpx/pull/3454
[416] https://github.com/STEllAR-GROUP/hpx/pull/3451
[417] https://github.com/STEllAR-GROUP/hpx/pull/3450
[418] https://github.com/STEllAR-GROUP/hpx/pull/3446
[419] https://github.com/STEllAR-GROUP/hpx/pull/3445
[420] https://github.com/STEllAR-GROUP/hpx/pull/3443
[421] https://github.com/STEllAR-GROUP/hpx/pull/3441

- PR #3439[422] - Use correct MPI CXX libraries for MPI parcelport
- PR #3436[423] - Add projection function to find_* (and fix very bad bug)
- PR #3435[424] - Fixing 1205
- PR #3434[425] - Fix threads cores
- PR #3433[426] - Add Heise Online to release announcement list
- PR #3432[427] - Don't track task dependencies for distributed runs
- PR #3431[428] - Circle CI setting changes for hpxMP
- PR #3430[429] - Fix unused params warning
- PR #3429[430] - One thread per core
- PR #3428[431] - This suppresses a deprecation warning that is being issued by MSVC 19.15.26726
- PR #3427[432] - Fixes #3426
- PR #3425[433] - Use source cache and workspace between job steps on CircleCI
- PR #3421[434] - Add CDash timing output to future overhead test (for graphs)
- PR #3420[435] - Add guided_pool_executor
- PR #3419[436] - Fix typo in CircleCI config
- PR #3418[437] - Add sphinx documentation
- PR #3415[438] - Scheduler NUMA hint and shared priority scheduler
- PR #3414[439] - Adding step to synchronize the APEX release
- PR #3413[440] - Fixing multiple defines of APEX_HAVE_HPX
- PR #3412[441] - Fixes linking with libhpx_wrap error with BSD and Windows based systems
- PR #3410[442] - Fix typo in CMakeLists.txt
- PR #3409[443] - Fix brackets and indentation in existing_performance_counters.qbk
- PR #3407[444] - Fix unused param and extra ; warnings emitted by gcc 8.x

[422] https://github.com/STEllAR-GROUP/hpx/pull/3439
[423] https://github.com/STEllAR-GROUP/hpx/pull/3436
[424] https://github.com/STEllAR-GROUP/hpx/pull/3435
[425] https://github.com/STEllAR-GROUP/hpx/pull/3434
[426] https://github.com/STEllAR-GROUP/hpx/pull/3433
[427] https://github.com/STEllAR-GROUP/hpx/pull/3432
[428] https://github.com/STEllAR-GROUP/hpx/pull/3431
[429] https://github.com/STEllAR-GROUP/hpx/pull/3430
[430] https://github.com/STEllAR-GROUP/hpx/pull/3429
[431] https://github.com/STEllAR-GROUP/hpx/pull/3428
[432] https://github.com/STEllAR-GROUP/hpx/pull/3427
[433] https://github.com/STEllAR-GROUP/hpx/pull/3425
[434] https://github.com/STEllAR-GROUP/hpx/pull/3421
[435] https://github.com/STEllAR-GROUP/hpx/pull/3420
[436] https://github.com/STEllAR-GROUP/hpx/pull/3419
[437] https://github.com/STEllAR-GROUP/hpx/pull/3418
[438] https://github.com/STEllAR-GROUP/hpx/pull/3415
[439] https://github.com/STEllAR-GROUP/hpx/pull/3414
[440] https://github.com/STEllAR-GROUP/hpx/pull/3413
[441] https://github.com/STEllAR-GROUP/hpx/pull/3412
[442] https://github.com/STEllAR-GROUP/hpx/pull/3410
[443] https://github.com/STEllAR-GROUP/hpx/pull/3409
[444] https://github.com/STEllAR-GROUP/hpx/pull/3407

- PR #3406[445] - Adding thread local allocator and use it for future shared states
- PR #3405[446] - Adding DHPX_HAVE_THREAD_LOCAL_STORAGE=ON to builds
- PR #3404[447] - fixing multiple difinition of main() in linux
- PR #3402[448] - Allow debug option to be enabled only for Linux systems with dynamic main on
- PR #3401[449] - Fix cuda_future_helper.h when compiling with C++11
- PR #3400[450] - Fix floating point exception scheduler_base idle backoff
- PR #3398[451] - Atomic future state
- PR #3397[452] - Fixing code for older gcc versions
- PR #3396[453] - Allowing to register thread event functions (start/stop/error)
- PR #3394[454] - Fix small mistake in primary_namespace_server.cpp
- PR #3393[455] - Explicitly instantiate configured schedulers
- PR #3392[456] - Add performance counters background overhead and background work duration
- PR #3391[457] - Adapt integration of HPXMP to latest build system changes
- PR #3390[458] - Make AGAS measurements optional
- PR #3389[459] - Fix deadlock during shutdown
- PR #3388[460] - Add several functionalities allowing to optimize synchronous action invocation
- PR #3387[461] - Add cmake option to opt out of fail-compile tests
- PR #3386[462] - Adding support for boost::container::small_vector to dataflow
- PR #3385[463] - Adds Debug option for hpx initializing from main
- PR #3384[464] - This hopefully fixes two tests that occasionally fail
- PR #3383[465] - Making sure thread local storage is enable for hpxMP
- PR #3382[466] - Fix usage of HPX_CAPTURE together with default value capture [=]
- PR #3381[467] - Replace undefined instantiations of uniform_int_distribution

[445] https://github.com/STEllAR-GROUP/hpx/pull/3406
[446] https://github.com/STEllAR-GROUP/hpx/pull/3405
[447] https://github.com/STEllAR-GROUP/hpx/pull/3404
[448] https://github.com/STEllAR-GROUP/hpx/pull/3402
[449] https://github.com/STEllAR-GROUP/hpx/pull/3401
[450] https://github.com/STEllAR-GROUP/hpx/pull/3400
[451] https://github.com/STEllAR-GROUP/hpx/pull/3398
[452] https://github.com/STEllAR-GROUP/hpx/pull/3397
[453] https://github.com/STEllAR-GROUP/hpx/pull/3396
[454] https://github.com/STEllAR-GROUP/hpx/pull/3394
[455] https://github.com/STEllAR-GROUP/hpx/pull/3393
[456] https://github.com/STEllAR-GROUP/hpx/pull/3392
[457] https://github.com/STEllAR-GROUP/hpx/pull/3391
[458] https://github.com/STEllAR-GROUP/hpx/pull/3390
[459] https://github.com/STEllAR-GROUP/hpx/pull/3389
[460] https://github.com/STEllAR-GROUP/hpx/pull/3388
[461] https://github.com/STEllAR-GROUP/hpx/pull/3387
[462] https://github.com/STEllAR-GROUP/hpx/pull/3386
[463] https://github.com/STEllAR-GROUP/hpx/pull/3385
[464] https://github.com/STEllAR-GROUP/hpx/pull/3384
[465] https://github.com/STEllAR-GROUP/hpx/pull/3383
[466] https://github.com/STEllAR-GROUP/hpx/pull/3382
[467] https://github.com/STEllAR-GROUP/hpx/pull/3381

- PR #3380[468] - Add missing semicolons to uses of HPX_COMPILER_FENCE
- PR #3379[469] - Fixing #3378
- PR #3377[470] - Adding build system support to integrate hpxmp into hpx at the user's machine
- PR #3375[471] - Replacing wrapper for __libc_start_main with main
- PR #3374[472] - Adds hpx_wrap to HPX_LINK_LIBRARIES which links only when specified.
- PR #3373[473] - Forcing cache settings in HPXConfig.cmake to guarantee updated values
- PR #3372[474] - Fix some more c++11 build problems
- PR #3371[475] - Adds HPX_LINKER_FLAGS to HPX applications without editing their source codes
- PR #3370[476] - util::format: add type_specifier<> specializations for %!s(MISSING) and %!l(MISSING)s
- PR #3369[477] - Adding configuration option to allow explicit disable of the new hpx_main feature on Linux
- PR #3368[478] - Updates doc with recent hpx_wrap implementation
- PR #3367[479] - Adds Mac OS implementation to hpx_main.hpp
- PR #3365[480] - Fix order of hpx libs in HPX_CONF_LIBRARIES.
- PR #3363[481] - Apex fixing null wrapper
- PR #3361[482] - Making sure all parcels get destroyed on an HPX thread (TCP pp)
- PR #3359[483] - Feature/improveerrorforcompiler
- PR #3357[484] - Static/dynamic executable implementation
- PR #3355[485] - Reverting changes introduced by #3283 as those make applications hang
- PR #3354[486] - Add external dependencies to HPX_LIBRARY_DIR
- PR #3353[487] - Fix libfabric tcp
- PR #3351[488] - Move obsolete header to tests directory.
- PR #3350[489] - Renaming two functions to avoid problem described in #3285
- PR #3349[490] - Make idle backoff exponential with maximum sleep time

468 https://github.com/STEllAR-GROUP/hpx/pull/3380
469 https://github.com/STEllAR-GROUP/hpx/pull/3379
470 https://github.com/STEllAR-GROUP/hpx/pull/3377
471 https://github.com/STEllAR-GROUP/hpx/pull/3375
472 https://github.com/STEllAR-GROUP/hpx/pull/3374
473 https://github.com/STEllAR-GROUP/hpx/pull/3373
474 https://github.com/STEllAR-GROUP/hpx/pull/3372
475 https://github.com/STEllAR-GROUP/hpx/pull/3371
476 https://github.com/STEllAR-GROUP/hpx/pull/3370
477 https://github.com/STEllAR-GROUP/hpx/pull/3369
478 https://github.com/STEllAR-GROUP/hpx/pull/3368
479 https://github.com/STEllAR-GROUP/hpx/pull/3367
480 https://github.com/STEllAR-GROUP/hpx/pull/3365
481 https://github.com/STEllAR-GROUP/hpx/pull/3363
482 https://github.com/STEllAR-GROUP/hpx/pull/3361
483 https://github.com/STEllAR-GROUP/hpx/pull/3359
484 https://github.com/STEllAR-GROUP/hpx/pull/3357
485 https://github.com/STEllAR-GROUP/hpx/pull/3355
486 https://github.com/STEllAR-GROUP/hpx/pull/3354
487 https://github.com/STEllAR-GROUP/hpx/pull/3353
488 https://github.com/STEllAR-GROUP/hpx/pull/3351
489 https://github.com/STEllAR-GROUP/hpx/pull/3350
490 https://github.com/STEllAR-GROUP/hpx/pull/3349

- PR #3347[491] - Replace *simple_component** with *component** in the Documentation
- PR #3346[492] - Fix CMakeLists.txt example in quick start
- PR #3345[493] - Fix automatic setting of HPX_MORE_THAN_64_THREADS
- PR #3344[494] - Reduce amount of information printed for unknown command line options
- PR #3343[495] - Safeguard HPX against destruction in global contexts
- PR #3341[496] - Allowing for all command line options to be used as configuration settings
- PR #3340[497] - Always convert inspect results to JUnit XML
- PR #3336[498] - Only run docker push on master on CircleCI
- PR #3335[499] - Update description of hpx.os_threads config parameter.
- PR #3334[500] - Making sure early logging settings don't get mixed with others
- PR #3333[501] - Update CMake links and versions in documentation
- PR #3332[502] - Add notes on target suffixes to CMake documentation
- PR #3331[503] - Add quickstart section to documentation
- PR #3330[504] - Rename resource_partitioner test to avoid conflicts with pseudodependencies
- PR #3328[505] - Making sure object is pinned while executing actions, even if action returns a future
- PR #3327[506] - Add missing std::forward to tuple.hpp
- PR #3326[507] - Make sure logging is up and running while modules are being discovered.
- PR #3324[508] - Replace C++14 overload of std::equal with C++11 code.
- PR #3323[509] - Fix a missing apex thread data (wrapper) initialization
- PR #3320[510] - Adding support for -std=c++2a (define *HPX_WITH_CXX2A=On*)
- PR #3319[511] - Replacing C++14 feature with equivalent C++11 code
- PR #3317[512] - Fix compilation with VS 15.7.1 and /std:c++latest
- PR #3316[513] - Fix includes for 1d_stencil_*_omp examples

[491] https://github.com/STEllAR-GROUP/hpx/pull/3347
[492] https://github.com/STEllAR-GROUP/hpx/pull/3346
[493] https://github.com/STEllAR-GROUP/hpx/pull/3345
[494] https://github.com/STEllAR-GROUP/hpx/pull/3344
[495] https://github.com/STEllAR-GROUP/hpx/pull/3343
[496] https://github.com/STEllAR-GROUP/hpx/pull/3341
[497] https://github.com/STEllAR-GROUP/hpx/pull/3340
[498] https://github.com/STEllAR-GROUP/hpx/pull/3336
[499] https://github.com/STEllAR-GROUP/hpx/pull/3335
[500] https://github.com/STEllAR-GROUP/hpx/pull/3334
[501] https://github.com/STEllAR-GROUP/hpx/pull/3333
[502] https://github.com/STEllAR-GROUP/hpx/pull/3332
[503] https://github.com/STEllAR-GROUP/hpx/pull/3331
[504] https://github.com/STEllAR-GROUP/hpx/pull/3330
[505] https://github.com/STEllAR-GROUP/hpx/pull/3328
[506] https://github.com/STEllAR-GROUP/hpx/pull/3327
[507] https://github.com/STEllAR-GROUP/hpx/pull/3326
[508] https://github.com/STEllAR-GROUP/hpx/pull/3324
[509] https://github.com/STEllAR-GROUP/hpx/pull/3323
[510] https://github.com/STEllAR-GROUP/hpx/pull/3320
[511] https://github.com/STEllAR-GROUP/hpx/pull/3319
[512] https://github.com/STEllAR-GROUP/hpx/pull/3317
[513] https://github.com/STEllAR-GROUP/hpx/pull/3316

- PR #3314[514] - Remove some unused parameter warnings
- PR #3313[515] - Fix pu-step and pu-offset command line options
- PR #3312[516] - Add conversion of inspect reports to JUnit XML
- PR #3311[517] - Fix escaping of closing braces in format specification syntax
- PR #3310[518] - Don't overwrite user settings with defaults in registration database
- PR #3309[519] - Fixing potential stack overflow for dataflow
- PR #3308[520] - This updates the .clang-format configuration file to utilize newer features
- PR #3306[521] - Marking migratable objects in their gid to allow not handling migration in AGAS
- PR #3305[522] - Add proper exception handling to run_as_hpx_thread
- PR #3303[523] - Changed std::rand to a better inbuilt PRNG Generator
- PR #3302[524] - All non-migratable (simple) components now encode their lva and component type in their gid
- PR #3301[525] - Add nullptr_t overloads to resource partitioner
- PR #3298[526] - Apex task wrapper memory bug
- PR #3295[527] - Fix mistakes after merge of CircleCI config
- PR #3294[528] - Fix partitioned vector include in partitioned_vector_find tests
- PR #3293[529] - Adding emplace support to promise and make_ready_future
- PR #3292[530] - Add new cuda kernel synchronization with hpx::future demo
- PR #3291[531] - Fixes #3290
- PR #3289[532] - Fixing Docker image creation
- PR #3288[533] - Avoid allocating shared state for wait_all
- PR #3287[534] - Fixing /scheduler/utilization/instantaneous performance counter
- PR #3286[535] - dataflow() and future::then() use sync policy where possible
- PR #3284[536] - Background thread can use relaxed atomics to manipulate thread state

[514] https://github.com/STEllAR-GROUP/hpx/pull/3314
[515] https://github.com/STEllAR-GROUP/hpx/pull/3313
[516] https://github.com/STEllAR-GROUP/hpx/pull/3312
[517] https://github.com/STEllAR-GROUP/hpx/pull/3311
[518] https://github.com/STEllAR-GROUP/hpx/pull/3310
[519] https://github.com/STEllAR-GROUP/hpx/pull/3309
[520] https://github.com/STEllAR-GROUP/hpx/pull/3308
[521] https://github.com/STEllAR-GROUP/hpx/pull/3306
[522] https://github.com/STEllAR-GROUP/hpx/pull/3305
[523] https://github.com/STEllAR-GROUP/hpx/pull/3303
[524] https://github.com/STEllAR-GROUP/hpx/pull/3302
[525] https://github.com/STEllAR-GROUP/hpx/pull/3301
[526] https://github.com/STEllAR-GROUP/hpx/pull/3298
[527] https://github.com/STEllAR-GROUP/hpx/pull/3295
[528] https://github.com/STEllAR-GROUP/hpx/pull/3294
[529] https://github.com/STEllAR-GROUP/hpx/pull/3293
[530] https://github.com/STEllAR-GROUP/hpx/pull/3292
[531] https://github.com/STEllAR-GROUP/hpx/pull/3291
[532] https://github.com/STEllAR-GROUP/hpx/pull/3289
[533] https://github.com/STEllAR-GROUP/hpx/pull/3288
[534] https://github.com/STEllAR-GROUP/hpx/pull/3287
[535] https://github.com/STEllAR-GROUP/hpx/pull/3286
[536] https://github.com/STEllAR-GROUP/hpx/pull/3284

- PR #3283[537] - Do not unwrap ready future
- PR #3282[538] - Fix virtual method override warnings in static schedulers
- PR #3281[539] - Disable set_area_membind_nodeset for OSX
- PR #3279[540] - Add two variations to the future_overhead benchmark
- PR #3278[541] - Fix circleci workspace
- PR #3277[542] - Support external plugins
- PR #3276[543] - Fix missing parenthesis in hello_compute.cu.
- PR #3274[544] - Reinit counters synchronously in reinit_counters test
- PR #3273[545] - Splitting tests to avoid compiler OOM
- PR #3271[546] - Remove leftover code from context_generic_context.hpp
- PR #3269[547] - Fix bulk_construct with count = 0
- PR #3268[548] - Replace constexpr with HPX_CXX14_CONSTEXPR and HPX_CONSTEXPR
- PR #3266[549] - Replace boost::format with custom sprintf-based implementation
- PR #3265[550] - Split parallel tests on CircleCI
- PR #3262[551] - Making sure documentation correctly links to source files
- PR #3261[552] - Apex refactoring fix rebind
- PR #3260[553] - Isolate performance counter parser into a separate TU
- PR #3256[554] - Post 1.1.0 version bumps
- PR #3254[555] - Adding trait for actions allowing to make runtime decision on whether to execute it directly
- PR #3253[556] - Bump minimal supported Boost to 1.58.0
- PR #3251[557] - Adds new feature: changing interval used in interval_timer (issue 3244)
- PR #3239[558] - Changing std::rand() to a better inbuilt PRNG generator.
- PR #3234[559] - Disable background thread when networking is off

[537] https://github.com/STEllAR-GROUP/hpx/pull/3283
[538] https://github.com/STEllAR-GROUP/hpx/pull/3282
[539] https://github.com/STEllAR-GROUP/hpx/pull/3281
[540] https://github.com/STEllAR-GROUP/hpx/pull/3279
[541] https://github.com/STEllAR-GROUP/hpx/pull/3278
[542] https://github.com/STEllAR-GROUP/hpx/pull/3277
[543] https://github.com/STEllAR-GROUP/hpx/pull/3276
[544] https://github.com/STEllAR-GROUP/hpx/pull/3274
[545] https://github.com/STEllAR-GROUP/hpx/pull/3273
[546] https://github.com/STEllAR-GROUP/hpx/pull/3271
[547] https://github.com/STEllAR-GROUP/hpx/pull/3269
[548] https://github.com/STEllAR-GROUP/hpx/pull/3268
[549] https://github.com/STEllAR-GROUP/hpx/pull/3266
[550] https://github.com/STEllAR-GROUP/hpx/pull/3265
[551] https://github.com/STEllAR-GROUP/hpx/pull/3262
[552] https://github.com/STEllAR-GROUP/hpx/pull/3261
[553] https://github.com/STEllAR-GROUP/hpx/pull/3260
[554] https://github.com/STEllAR-GROUP/hpx/pull/3256
[555] https://github.com/STEllAR-GROUP/hpx/pull/3254
[556] https://github.com/STEllAR-GROUP/hpx/pull/3253
[557] https://github.com/STEllAR-GROUP/hpx/pull/3251
[558] https://github.com/STEllAR-GROUP/hpx/pull/3239
[559] https://github.com/STEllAR-GROUP/hpx/pull/3234

- PR #3232[560] - Clean up suspension tests
- PR #3230[561] - Add optional scheduler mode parameter to create_thread_pool function
- PR #3228[562] - Allow suspension also on static schedulers
- PR #3163[563] - libfabric parcelport w/o HPX_PARCELPORT_LIBFABRIC_ENDPOINT_RDM
- PR #3036[564] - Switching to CircleCI 2.0

2.10.4 *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[565] and/or MPI[566].
- 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[567]. 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[568] 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[569].
- 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.

560 https://github.com/STEllAR-GROUP/hpx/pull/3232
561 https://github.com/STEllAR-GROUP/hpx/pull/3230
562 https://github.com/STEllAR-GROUP/hpx/pull/3228
563 https://github.com/STEllAR-GROUP/hpx/pull/3163
564 https://github.com/STEllAR-GROUP/hpx/pull/3036
565 https://openmp.org/wp/
566 https://en.wikipedia.org/wiki/Message_Passing_Interface
567 http://www.open-std.org/jtc1/sc22/wg21
568 https://www.cmake.org
569 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 the compatibility option `-DHPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT=On` on the CMake[570] 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 to defer the decision on what launch policy to use to the point of execution. This policy is initialized with a function (object) that – when invoked – is expected to produce the desired launch policy.

Breaking changes

- We have dropped support for the gcc compiler version V4.8. The minimal gcc version we now test on is gcc V4.9. The minimally required version of CMake[571] 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.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #3250[572] - Apex refactoring with guids
- PR #3249[573] - Updating People.qbk

[570] https://www.cmake.org
[571] https://www.cmake.org
[572] https://github.com/STEllAR-GROUP/hpx/pull/3250
[573] https://github.com/STEllAR-GROUP/hpx/pull/3249

- PR #3246[574] - Assorted fixes for CUDA
- PR #3245[575] - Apex refactoring with guids
- PR #3242[576] - Modify task counting in thread_queue.hpp
- PR #3240[577] - Fixed typos
- PR #3238[578] - Readding accidently removed std::abort
- PR #3237[579] - Adding Pipeline example
- PR #3236[580] - Fixing memory_block
- PR #3233[581] - Make schedule_thread take suspended threads into account
- Issue #3226[582] - memory_block is breaking, signaling SIGSEGV on a thread on creation and freeing
- PR #3225[583] - Applying quick fix for hwloc-2.0
- Issue #3224[584] - HPX counters crashing the application
- PR #3223[585] - Fix returns when setting config entries
- Issue #3222[586] - Errors linking libhpx.so
- Issue #3221[587] - HPX on Mac OS X with HWLoc 2.0.0 fails to run
- PR #3216[588] - Reorder a variadic array to satisfy VS 2017 15.6
- PR #3214[589] - Changed prerequisites.qbk to avoid confusion while building boost
- PR #3213[590] - Relax locks for thread suspension to avoid holding locks when yielding
- PR #3212[591] - Fix check in sequenced_executor test
- PR #3211[592] - Use preinit_array to set argc/argv in init_globally example
- PR #3210[593] - Adapted parallel::{search | search_n} for Ranges TS (see #1668)
- PR #3209[594] - Fix locking problems during shutdown
- Issue #3208[595] - init_globally throwing a run-time error
- PR #3206[596] - Addition of new arithmetic performance counter “Count”

574 https://github.com/STEllAR-GROUP/hpx/pull/3246
575 https://github.com/STEllAR-GROUP/hpx/pull/3245
576 https://github.com/STEllAR-GROUP/hpx/pull/3242
577 https://github.com/STEllAR-GROUP/hpx/pull/3240
578 https://github.com/STEllAR-GROUP/hpx/pull/3238
579 https://github.com/STEllAR-GROUP/hpx/pull/3237
580 https://github.com/STEllAR-GROUP/hpx/pull/3236
581 https://github.com/STEllAR-GROUP/hpx/pull/3233
582 https://github.com/STEllAR-GROUP/hpx/issues/3226
583 https://github.com/STEllAR-GROUP/hpx/pull/3225
584 https://github.com/STEllAR-GROUP/hpx/issues/3224
585 https://github.com/STEllAR-GROUP/hpx/pull/3223
586 https://github.com/STEllAR-GROUP/hpx/issues/3222
587 https://github.com/STEllAR-GROUP/hpx/issues/3221
588 https://github.com/STEllAR-GROUP/hpx/pull/3216
589 https://github.com/STEllAR-GROUP/hpx/pull/3214
590 https://github.com/STEllAR-GROUP/hpx/pull/3213
591 https://github.com/STEllAR-GROUP/hpx/pull/3212
592 https://github.com/STEllAR-GROUP/hpx/pull/3211
593 https://github.com/STEllAR-GROUP/hpx/pull/3210
594 https://github.com/STEllAR-GROUP/hpx/pull/3209
595 https://github.com/STEllAR-GROUP/hpx/issues/3208
596 https://github.com/STEllAR-GROUP/hpx/pull/3206

- PR #3205[597] - Fixing return type calculation for bulk_then_execute
- PR #3204[598] - Changing std::rand() to a better inbuilt PRNG generator
- PR #3203[599] - Resolving problems during shutdown for VS2015
- PR #3202[600] - Making sure resource partitioner is not accessed if its not valid
- PR #3201[601] - Fixing optional::swap
- Issue #3200[602] - hpx::util::optional fails
- PR #3199[603] - Fix sliding_semaphore test
- PR #3198[604] - Set pre_main status before launching run_helper
- PR #3197[605] - Update README.rst
- PR #3194[606] - parallel::{fill|fill_n} updated for Ranges TS
- PR #3193[607] - Updating Runtime.cpp by adding correct description of Performance counters during register
- PR #3191[608] - Fix sliding_semaphore_2338 test
- PR #3190[609] - Topology improvements
- PR #3189[610] - Deleting one include of median from BOOST library to arithmetics_counter file
- PR #3188[611] - Optionally disable printing of diagnostics during terminate
- PR #3187[612] - Suppressing cmake warning issued by cmake > V3.11
- PR #3185[613] - Remove unused scoped_unlock, unlock_guard_try
- PR #3184[614] - Fix nqueen example
- PR #3183[615] - Add runtime start/stop, resume/suspend and OpenMP benchmarks
- Issue #3182[616] - bulk_then_execute has unexpected return type/does not compile
- Issue #3181[617] - hwloc 2.0 breaks topo class and cannot be used
- Issue #3180[618] - Schedulers that don't support suspend/resume are unusable
- PR #3179[619] - Various minor changes to support FLeCSI

597 https://github.com/STEllAR-GROUP/hpx/pull/3205
598 https://github.com/STEllAR-GROUP/hpx/pull/3204
599 https://github.com/STEllAR-GROUP/hpx/pull/3203
600 https://github.com/STEllAR-GROUP/hpx/pull/3202
601 https://github.com/STEllAR-GROUP/hpx/pull/3201
602 https://github.com/STEllAR-GROUP/hpx/issues/3200
603 https://github.com/STEllAR-GROUP/hpx/pull/3199
604 https://github.com/STEllAR-GROUP/hpx/pull/3198
605 https://github.com/STEllAR-GROUP/hpx/pull/3197
606 https://github.com/STEllAR-GROUP/hpx/pull/3194
607 https://github.com/STEllAR-GROUP/hpx/pull/3193
608 https://github.com/STEllAR-GROUP/hpx/pull/3191
609 https://github.com/STEllAR-GROUP/hpx/pull/3190
610 https://github.com/STEllAR-GROUP/hpx/pull/3189
611 https://github.com/STEllAR-GROUP/hpx/pull/3188
612 https://github.com/STEllAR-GROUP/hpx/pull/3187
613 https://github.com/STEllAR-GROUP/hpx/pull/3185
614 https://github.com/STEllAR-GROUP/hpx/pull/3184
615 https://github.com/STEllAR-GROUP/hpx/pull/3183
616 https://github.com/STEllAR-GROUP/hpx/issues/3182
617 https://github.com/STEllAR-GROUP/hpx/issues/3181
618 https://github.com/STEllAR-GROUP/hpx/issues/3180
619 https://github.com/STEllAR-GROUP/hpx/pull/3179

- PR #3178[620] - Fix #3124
- PR #3177[621] - Removed allgather
- PR #3176[622] - Fixed Documentation for "using_hpx_pkgconfig"
- PR #3174[623] - Add hpx::iostreams::ostream overload to format_to
- PR #3172[624] - Fix lifo queue backend
- PR #3171[625] - adding the missing unset() function to cpu_mask() for case of more than 64 threads
- PR #3170[626] - Add cmake flag -DHPX_WITH_FAULT_TOLERANCE=ON (OFF by default)
- PR #3169[627] - Adapted parallel::{count|count_if} for Ranges TS (see #1668)
- PR #3168[628] - Changing used namespace for seq execution policy
- Issue #3167[629] - Update GSoC projects
- Issue #3166[630] - Application (Octotiger) gets stuck on hpx::finalize when only using one thread
- Issue #3165[631] - Compilation of parallel algorithms with HPX_WITH_DATAPAR is broken
- PR #3164[632] - Fixing component migration
- PR #3162[633] - regex_from_pattern: escape regex special characters to avoid misinterpretation
- Issue #3161[634] - Building HPX with hwloc 2.0.0 fails
- PR #3160[635] - Fixing the handling of quoted command line arguments.
- PR #3158[636] - Fixing a race with timed suspension (second attempt)
- PR #3157[637] - Revert "Fixing a race with timed suspension"
- PR #3156[638] - Fixing serialization of classes with incompatible serialize signature
- PR #3154[639] - More refactorings based on clang-tidy reports
- PR #3153[640] - Fixing a race with timed suspension
- PR #3152[641] - Documentation for runtime suspension
- PR #3151[642] - Use small_vector only from boost version 1.59 onwards

[620] https://github.com/STEllAR-GROUP/hpx/pull/3178
[621] https://github.com/STEllAR-GROUP/hpx/pull/3177
[622] https://github.com/STEllAR-GROUP/hpx/pull/3176
[623] https://github.com/STEllAR-GROUP/hpx/pull/3174
[624] https://github.com/STEllAR-GROUP/hpx/pull/3172
[625] https://github.com/STEllAR-GROUP/hpx/pull/3171
[626] https://github.com/STEllAR-GROUP/hpx/pull/3170
[627] https://github.com/STEllAR-GROUP/hpx/pull/3169
[628] https://github.com/STEllAR-GROUP/hpx/pull/3168
[629] https://github.com/STEllAR-GROUP/hpx/issues/3167
[630] https://github.com/STEllAR-GROUP/hpx/issues/3166
[631] https://github.com/STEllAR-GROUP/hpx/issues/3165
[632] https://github.com/STEllAR-GROUP/hpx/pull/3164
[633] https://github.com/STEllAR-GROUP/hpx/pull/3162
[634] https://github.com/STEllAR-GROUP/hpx/issues/3161
[635] https://github.com/STEllAR-GROUP/hpx/pull/3160
[636] https://github.com/STEllAR-GROUP/hpx/pull/3158
[637] https://github.com/STEllAR-GROUP/hpx/pull/3157
[638] https://github.com/STEllAR-GROUP/hpx/pull/3156
[639] https://github.com/STEllAR-GROUP/hpx/pull/3154
[640] https://github.com/STEllAR-GROUP/hpx/pull/3153
[641] https://github.com/STEllAR-GROUP/hpx/pull/3152
[642] https://github.com/STEllAR-GROUP/hpx/pull/3151

- PR #3150[643] - Avoiding more stack overflows
- PR #3148[644] - Refactoring component_base and base_action/transfer_base_action
- PR #3147[645] - Move yield_while out of detail namespace and into own file
- PR #3145[646] - Remove a leftover of the cxx11 std array cleanup
- PR #3144[647] - Minor changes to how actions are executed
- PR #3143[648] - Fix stack overhead
- PR #3142[649] - Fix typo in config.hpp
- PR #3141[650] - Fixing small_vector compatibility with older boost version
- PR #3140[651] - is_heap_text fix
- Issue #3139[652] - Error in is_heap_tests.hpp
- PR #3138[653] - Partially reverting #3126
- PR #3137[654] - Suspend speedup
- PR #3136[655] - Revert "Fixing #2325"
- PR #3135[656] - Improving destruction of threads
- Issue #3134[657] - HPX_SERIALIZATION_SPLIT_FREE does not stop compiler from looking for serialize() method
- PR #3133[658] - Make hwloc compulsory
- PR #3132[659] - Update CXX14 constexpr feature test
- PR #3131[660] - Fixing #2325
- PR #3130[661] - Avoid completion handler allocation
- PR #3129[662] - Suspend runtime
- PR #3128[663] - Make docbook dtd and xsl path names consistent
- PR #3127[664] - Add hpx::start nullptr overloads
- PR #3126[665] - Cleaning up coroutine implementation

643 https://github.com/STEllAR-GROUP/hpx/pull/3150
644 https://github.com/STEllAR-GROUP/hpx/pull/3148
645 https://github.com/STEllAR-GROUP/hpx/pull/3147
646 https://github.com/STEllAR-GROUP/hpx/pull/3145
647 https://github.com/STEllAR-GROUP/hpx/pull/3144
648 https://github.com/STEllAR-GROUP/hpx/pull/3143
649 https://github.com/STEllAR-GROUP/hpx/pull/3142
650 https://github.com/STEllAR-GROUP/hpx/pull/3141
651 https://github.com/STEllAR-GROUP/hpx/pull/3140
652 https://github.com/STEllAR-GROUP/hpx/issues/3139
653 https://github.com/STEllAR-GROUP/hpx/pull/3138
654 https://github.com/STEllAR-GROUP/hpx/pull/3137
655 https://github.com/STEllAR-GROUP/hpx/pull/3136
656 https://github.com/STEllAR-GROUP/hpx/pull/3135
657 https://github.com/STEllAR-GROUP/hpx/issues/3134
658 https://github.com/STEllAR-GROUP/hpx/pull/3133
659 https://github.com/STEllAR-GROUP/hpx/pull/3132
660 https://github.com/STEllAR-GROUP/hpx/pull/3131
661 https://github.com/STEllAR-GROUP/hpx/pull/3130
662 https://github.com/STEllAR-GROUP/hpx/pull/3129
663 https://github.com/STEllAR-GROUP/hpx/pull/3128
664 https://github.com/STEllAR-GROUP/hpx/pull/3127
665 https://github.com/STEllAR-GROUP/hpx/pull/3126

- PR #3125[666] - Replacing nullptr with hpx::threads::invalid_thread_id
- Issue #3124[667] - Add hello_world_component to CI builds
- PR #3123[668] - Add new constructor.
- PR #3122[669] - Fixing #3121
- Issue #3121[670] - HPX_SMT_PAUSE is broken on non-x86 platforms when __GNUC__ is defined
- PR #3120[671] - Don't use boost::intrusive_ptr for thread_id_type
- PR #3119[672] - Disable default executor compatibility with V1 executors
- PR #3118[673] - Adding performance_counter::reinit to allow for dynamically changing counter sets
- PR #3117[674] - Replace uses of boost/experimental::optional with util::optional
- PR #3116[675] - Moving background thread APEX timer #2980
- PR #3115[676] - Fixing race condition in channel test
- PR #3114[677] - Avoid using util::function for thread function wrappers
- PR #3113[678] - cmake V3.10.2 has changed the variable names used for MPI
- PR #3112[679] - Minor fixes to exclusive_scan algorithm
- PR #3111[680] - Revert "fix detection of cxx11_std_atomic"
- PR #3110[681] - Suspend thread pool
- PR #3109[682] - Fixing thread scheduling when yielding a thread id
- PR #3108[683] - Revert "Suspend thread pool"
- PR #3107[684] - Remove UB from thread::id relational operators
- PR #3106[685] - Add cmake test for std::decay_t to fix cuda build
- PR #3105[686] - Fixing refcount for async traversal frame
- PR #3104[687] - Local execution of direct actions is now actually performed directly
- PR #3103[688] - Adding support for generic counter_raw_values performance counter type

666 https://github.com/STEllAR-GROUP/hpx/pull/3125
667 https://github.com/STEllAR-GROUP/hpx/issues/3124
668 https://github.com/STEllAR-GROUP/hpx/pull/3123
669 https://github.com/STEllAR-GROUP/hpx/pull/3122
670 https://github.com/STEllAR-GROUP/hpx/issues/3121
671 https://github.com/STEllAR-GROUP/hpx/pull/3120
672 https://github.com/STEllAR-GROUP/hpx/pull/3119
673 https://github.com/STEllAR-GROUP/hpx/pull/3118
674 https://github.com/STEllAR-GROUP/hpx/pull/3117
675 https://github.com/STEllAR-GROUP/hpx/pull/3116
676 https://github.com/STEllAR-GROUP/hpx/pull/3115
677 https://github.com/STEllAR-GROUP/hpx/pull/3114
678 https://github.com/STEllAR-GROUP/hpx/pull/3113
679 https://github.com/STEllAR-GROUP/hpx/pull/3112
680 https://github.com/STEllAR-GROUP/hpx/pull/3111
681 https://github.com/STEllAR-GROUP/hpx/pull/3110
682 https://github.com/STEllAR-GROUP/hpx/pull/3109
683 https://github.com/STEllAR-GROUP/hpx/pull/3108
684 https://github.com/STEllAR-GROUP/hpx/pull/3107
685 https://github.com/STEllAR-GROUP/hpx/pull/3106
686 https://github.com/STEllAR-GROUP/hpx/pull/3105
687 https://github.com/STEllAR-GROUP/hpx/pull/3104
688 https://github.com/STEllAR-GROUP/hpx/pull/3103

- Issue #3102[689] - Introduce generic performance counter type returning an array of values
- PR #3101[690] - Revert "Adapting stack overhead limit for gcc 4.9"
- PR #3100[691] - Fix #3068 (condition_variable deadlock)
- PR #3099[692] - Fixing lock held during suspension in papi counter component
- PR #3098[693] - Unbreak broadcast_wait_for_2822 test
- PR #3097[694] - Adapting stack overhead limit for gcc 4.9
- PR #3096[695] - fix detection of cxx11_std_atomic
- PR #3095[696] - Add ciso646 header to get _LIBCPP_VERSION for testing inplace merge
- PR #3094[697] - Relax atomic operations on performance counter values
- PR #3093[698] - Short-circuit all_of/any_of/none_of instantiations
- PR #3092[699] - Take advantage of C++14 lambda capture initialization syntax, where possible
- PR #3091[700] - Remove more references to Boost from logging code
- PR #3090[701] - Unify use of yield/yield_k
- PR #3089[702] - Fix a strange thing in parallel::detail::handle_exception. (Fix #2834.)
- Issue #3088[703] - A strange thing in parallel::sort.
- PR #3087[704] - Fixing assertion in default_distribution_policy
- PR #3086[705] - Implement parallel::remove and parallel::remove_if
- PR #3085[706] - Addressing breaking changes in Boost V1.66
- PR #3084[707] - Ignore build warnings round 2
- PR #3083[708] - Fix typo HPX_WITH_MM_PREFECTH
- PR #3081[709] - Pre-decay template arguments early
- PR #3080[710] - Suspend thread pool
- PR #3079[711] - Ignore build warnings

689 https://github.com/STEllAR-GROUP/hpx/issues/3102
690 https://github.com/STEllAR-GROUP/hpx/pull/3101
691 https://github.com/STEllAR-GROUP/hpx/pull/3100
692 https://github.com/STEllAR-GROUP/hpx/pull/3099
693 https://github.com/STEllAR-GROUP/hpx/pull/3098
694 https://github.com/STEllAR-GROUP/hpx/pull/3097
695 https://github.com/STEllAR-GROUP/hpx/pull/3096
696 https://github.com/STEllAR-GROUP/hpx/pull/3095
697 https://github.com/STEllAR-GROUP/hpx/pull/3094
698 https://github.com/STEllAR-GROUP/hpx/pull/3093
699 https://github.com/STEllAR-GROUP/hpx/pull/3092
700 https://github.com/STEllAR-GROUP/hpx/pull/3091
701 https://github.com/STEllAR-GROUP/hpx/pull/3090
702 https://github.com/STEllAR-GROUP/hpx/pull/3089
703 https://github.com/STEllAR-GROUP/hpx/issues/3088
704 https://github.com/STEllAR-GROUP/hpx/pull/3087
705 https://github.com/STEllAR-GROUP/hpx/pull/3086
706 https://github.com/STEllAR-GROUP/hpx/pull/3085
707 https://github.com/STEllAR-GROUP/hpx/pull/3084
708 https://github.com/STEllAR-GROUP/hpx/pull/3083
709 https://github.com/STEllAR-GROUP/hpx/pull/3081
710 https://github.com/STEllAR-GROUP/hpx/pull/3080
711 https://github.com/STEllAR-GROUP/hpx/pull/3079

- PR #3078[712] - Don't test inplace_merge with libc++
- PR #3076[713] - Fixing 3075: Part 1
- PR #3074[714] - Fix more build warnings
- PR #3073[715] - Suspend thread cleanup
- PR #3072[716] - Change existing symbol_namespace::iterate to return all data instead of invoking a callback
- PR #3071[717] - Fixing pack_traversal_async test
- PR #3070[718] - Fix dynamic_counters_loaded_1508 test by adding dependency to memory_component
- PR #3069[719] - Fix scheduling loop exit
- Issue #3068[720] - hpx::lcos::condition_variable could be suspect to deadlocks
- PR #3067[721] - #ifdef out random_shuffle deprecated in later c++
- PR #3066[722] - Make coalescing test depend on coalescing library to ensure it gets built
- PR #3065[723] - Workaround for minimal_timed_async_executor_test compilation failures, attempts to copy a deferred call (in unevaluated context)
- PR #3064[724] - Fixing wrong condition in wrapper_heap
- PR #3062[725] - Fix exception handling for execution::seq
- PR #3061[726] - Adapt MSVC C++ mode handling to VS15.5
- PR #3060[727] - Fix compiler problem in MSVC release mode
- PR #3059[728] - Fixing #2931
- Issue #3058[729] - minimal_timed_async_executor_test_exe fails to compile on master (d6f505c)
- PR #3057[730] - Fix stable_merge_2964 compilation problems
- PR #3056[731] - Fix some build warnings caused by unused variables/unnecessary tests
- PR #3055[732] - Update documentation for running tests
- Issue #3054[733] - Assertion failure when using bulk hpx::**new_** in asynchronous mode
- PR #3052[734] - Do not bind test running to cmake test build rule

[712] https://github.com/STEllAR-GROUP/hpx/pull/3078
[713] https://github.com/STEllAR-GROUP/hpx/pull/3076
[714] https://github.com/STEllAR-GROUP/hpx/pull/3074
[715] https://github.com/STEllAR-GROUP/hpx/pull/3073
[716] https://github.com/STEllAR-GROUP/hpx/pull/3072
[717] https://github.com/STEllAR-GROUP/hpx/pull/3071
[718] https://github.com/STEllAR-GROUP/hpx/pull/3070
[719] https://github.com/STEllAR-GROUP/hpx/pull/3069
[720] https://github.com/STEllAR-GROUP/hpx/issues/3068
[721] https://github.com/STEllAR-GROUP/hpx/pull/3067
[722] https://github.com/STEllAR-GROUP/hpx/pull/3066
[723] https://github.com/STEllAR-GROUP/hpx/pull/3065
[724] https://github.com/STEllAR-GROUP/hpx/pull/3064
[725] https://github.com/STEllAR-GROUP/hpx/pull/3062
[726] https://github.com/STEllAR-GROUP/hpx/pull/3061
[727] https://github.com/STEllAR-GROUP/hpx/pull/3060
[728] https://github.com/STEllAR-GROUP/hpx/pull/3059
[729] https://github.com/STEllAR-GROUP/hpx/issues/3058
[730] https://github.com/STEllAR-GROUP/hpx/pull/3057
[731] https://github.com/STEllAR-GROUP/hpx/pull/3056
[732] https://github.com/STEllAR-GROUP/hpx/pull/3055
[733] https://github.com/STEllAR-GROUP/hpx/issues/3054
[734] https://github.com/STEllAR-GROUP/hpx/pull/3052

- PR #3051[735] - Fix HPX-Qt interaction in Qt example.
- Issue #3048[736] - nqueen example fails occasionally
- PR #3047[737] - Fixing #3044
- PR #3046[738] - Add OS thread suspension
- PR #3042[739] - PyCicle - first attempt at a build toold for checking PR's
- PR #3041[740] - Fix a problem about asynchronous execution of parallel::merge and parallel::partition.
- PR #3040[741] - Fix a mistake about exception handling in asynchronous execution of scan_partitioner.
- PR #3039[742] - Consistently use executors to schedule work
- PR #3038[743] - Fixing local direct function execution and lambda actions perfect forwarding
- PR #3035[744] - Make parallel unit test names match build target/folder names
- PR #3033[745] - Fix setting of default build type
- Issue #3032[746] - Fix partitioner arg copy found in #2982
- Issue #3031[747] - Errors linking libhpx.so due to missing references (master branch, commit 6679a8882)
- PR #3030[748] - Revert "implement executor then interface with && forwarding reference"
- PR #3029[749] - Run CI inspect checks before building
- PR #3028[750] - Added range version of parallel::move
- Issue #3027[751] - Implement all scheduling APIs in terms of executors
- PR #3026[752] - implement executor then interface with && forwarding reference
- PR #3025[753] - Fix typo unitialized to uninitialized
- PR #3024[754] - Inspect fixes
- PR #3023[755] - P0356 Simplified partial function application
- PR #3022[756] - Master fixes
- PR #3021[757] - Segfault fix

[735] https://github.com/STEllAR-GROUP/hpx/pull/3051
[736] https://github.com/STEllAR-GROUP/hpx/issues/3048
[737] https://github.com/STEllAR-GROUP/hpx/pull/3047
[738] https://github.com/STEllAR-GROUP/hpx/pull/3046
[739] https://github.com/STEllAR-GROUP/hpx/pull/3042
[740] https://github.com/STEllAR-GROUP/hpx/pull/3041
[741] https://github.com/STEllAR-GROUP/hpx/pull/3040
[742] https://github.com/STEllAR-GROUP/hpx/pull/3039
[743] https://github.com/STEllAR-GROUP/hpx/pull/3038
[744] https://github.com/STEllAR-GROUP/hpx/pull/3035
[745] https://github.com/STEllAR-GROUP/hpx/pull/3033
[746] https://github.com/STEllAR-GROUP/hpx/issues/3032
[747] https://github.com/STEllAR-GROUP/hpx/issues/3031
[748] https://github.com/STEllAR-GROUP/hpx/pull/3030
[749] https://github.com/STEllAR-GROUP/hpx/pull/3029
[750] https://github.com/STEllAR-GROUP/hpx/pull/3028
[751] https://github.com/STEllAR-GROUP/hpx/issues/3027
[752] https://github.com/STEllAR-GROUP/hpx/pull/3026
[753] https://github.com/STEllAR-GROUP/hpx/pull/3025
[754] https://github.com/STEllAR-GROUP/hpx/pull/3024
[755] https://github.com/STEllAR-GROUP/hpx/pull/3023
[756] https://github.com/STEllAR-GROUP/hpx/pull/3022
[757] https://github.com/STEllAR-GROUP/hpx/pull/3021

- PR #3020[758] - Disable command-line aliasing for applications that use user_main
- PR #3019[759] - Adding enable_elasticity option to pool configuration
- PR #3018[760] - Fix stack overflow detection configuration in header files
- PR #3017[761] - Speed up local action execution
- PR #3016[762] - Unify stack-overflow detection options, remove reference to libsigsegv
- PR #3015[763] - Speeding up accessing the resource partitioner and the topology info
- Issue #3014[764] - HPX does not compile on POWER8 with gcc 5.4
- Issue #3013[765] - hello_world occasionally prints multiple lines from a single OS-thread
- PR #3012[766] - Silence warning about casting away qualifiers in itt_notify.hpp
- PR #3011[767] - Fix cpuset leak in hwloc_topology_info.cpp
- PR #3010[768] - Remove useless decay_copy
- PR #3009[769] - Fixing 2996
- PR #3008[770] - Remove unused internal function
- PR #3007[771] - Fixing wrapper_heap alignment problems
- Issue #3006[772] - hwloc memory leak
- PR #3004[773] - Silence C4251 (needs to have dll-interface) for future_data_void
- Issue #3003[774] - Suspension of runtime
- PR #3001[775] - Attempting to avoid data races in async_traversal while evaluating dataflow()
- PR #3000[776] - Adding hpx::util::optional as a first step to replace experimental::optional
- PR #2998[777] - Cleanup up and Fixing component creation and deletion
- Issue #2996[778] - Build fails with HPX_WITH_HWLOC=OFF
- PR #2995[779] - Push more future_data functionality to source file
- PR #2994[780] - WIP: Fix throttle test

[758] https://github.com/STEllAR-GROUP/hpx/pull/3020
[759] https://github.com/STEllAR-GROUP/hpx/pull/3019
[760] https://github.com/STEllAR-GROUP/hpx/pull/3018
[761] https://github.com/STEllAR-GROUP/hpx/pull/3017
[762] https://github.com/STEllAR-GROUP/hpx/pull/3016
[763] https://github.com/STEllAR-GROUP/hpx/pull/3015
[764] https://github.com/STEllAR-GROUP/hpx/issues/3014
[765] https://github.com/STEllAR-GROUP/hpx/issues/3013
[766] https://github.com/STEllAR-GROUP/hpx/pull/3012
[767] https://github.com/STEllAR-GROUP/hpx/pull/3011
[768] https://github.com/STEllAR-GROUP/hpx/pull/3010
[769] https://github.com/STEllAR-GROUP/hpx/pull/3009
[770] https://github.com/STEllAR-GROUP/hpx/pull/3008
[771] https://github.com/STEllAR-GROUP/hpx/pull/3007
[772] https://github.com/STEllAR-GROUP/hpx/issues/3006
[773] https://github.com/STEllAR-GROUP/hpx/pull/3004
[774] https://github.com/STEllAR-GROUP/hpx/issues/3003
[775] https://github.com/STEllAR-GROUP/hpx/pull/3001
[776] https://github.com/STEllAR-GROUP/hpx/pull/3000
[777] https://github.com/STEllAR-GROUP/hpx/pull/2998
[778] https://github.com/STEllAR-GROUP/hpx/issues/2996
[779] https://github.com/STEllAR-GROUP/hpx/pull/2995
[780] https://github.com/STEllAR-GROUP/hpx/pull/2994

- PR #2993[781] - Making sure –hpx:help does not throw for required (but missing) arguments
- PR #2992[782] - Adding non-blocking (on destruction) service executors
- Issue #2991[783] - run_as_os_thread locks up
- Issue #2990[784] - –help will not work until all required options are provided
- PR #2989[785] - Improve error messages caused by misuse of dataflow
- PR #2988[786] - Improve error messages caused by misuse of .then
- Issue #2987[787] - stack overflow detection producing false positives
- PR #2986[788] - Deduplicate non-dependent thread_info logging types
- PR #2985[789] - Adapted parallel::{all_of|any_of|none_of} for Ranges TS (see #1668)
- PR #2984[790] - Refactor one_size_heap code to simplify code
- PR #2983[791] - Fixing local_new_component
- PR #2982[792] - Clang tidy
- PR #2981[793] - Simplify allocator rebinding in pack traversal
- PR #2979[794] - Fixing integer overflows
- PR #2978[795] - Implement parallel::inplace_merge
- Issue #2977[796] - Make hwloc compulsory instead of optional
- PR #2976[797] - Making sure client_base instance that registered the component does not unregister it when being destructed
- PR #2975[798] - Change version of pulled APEX to master
- PR #2974[799] - Fix domain not being freed at the end of scheduling loop
- PR #2973[800] - Fix small typos
- PR #2972[801] - Adding uintstd.h header
- PR #2971[802] - Fall back to creating local components using local_new
- PR #2970[803] - Improve is_tuple_like trait

781 https://github.com/STEllAR-GROUP/hpx/pull/2993
782 https://github.com/STEllAR-GROUP/hpx/pull/2992
783 https://github.com/STEllAR-GROUP/hpx/issues/2991
784 https://github.com/STEllAR-GROUP/hpx/issues/2990
785 https://github.com/STEllAR-GROUP/hpx/pull/2989
786 https://github.com/STEllAR-GROUP/hpx/pull/2988
787 https://github.com/STEllAR-GROUP/hpx/issues/2987
788 https://github.com/STEllAR-GROUP/hpx/pull/2986
789 https://github.com/STEllAR-GROUP/hpx/pull/2985
790 https://github.com/STEllAR-GROUP/hpx/pull/2984
791 https://github.com/STEllAR-GROUP/hpx/pull/2983
792 https://github.com/STEllAR-GROUP/hpx/pull/2982
793 https://github.com/STEllAR-GROUP/hpx/pull/2981
794 https://github.com/STEllAR-GROUP/hpx/pull/2979
795 https://github.com/STEllAR-GROUP/hpx/pull/2978
796 https://github.com/STEllAR-GROUP/hpx/issues/2977
797 https://github.com/STEllAR-GROUP/hpx/pull/2976
798 https://github.com/STEllAR-GROUP/hpx/pull/2975
799 https://github.com/STEllAR-GROUP/hpx/pull/2974
800 https://github.com/STEllAR-GROUP/hpx/pull/2973
801 https://github.com/STEllAR-GROUP/hpx/pull/2972
802 https://github.com/STEllAR-GROUP/hpx/pull/2971
803 https://github.com/STEllAR-GROUP/hpx/pull/2970

- PR #2969[804] - Fix HPX_WITH_MORE_THAN_64_THREADS default value
- PR #2968[805] - Cleaning up dataflow overload set
- PR #2967[806] - Make parallel::merge is stable. (Fix #2964.)
- PR #2966[807] - Fixing a couple of held locks during exception handling
- PR #2965[808] - Adding missing #include
- Issue #2964[809] - parallel merge is not stable
- PR #2963[810] - Making sure any function object passed to dataflow is released after being invoked
- PR #2962[811] - Partially reverting #2891
- PR #2961[812] - Attempt to fix the gcc 4.9 problem with the async pack traversal
- Issue #2959[813] - Program terminates during error handling
- Issue #2958[814] - HPX_PLAIN_ACTION breaks due to missing include
- PR #2957[815] - Fixing errors generated by mixing different attribute syntaxes
- Issue #2956[816] - Mixing attribute syntaxes leads to compiler errors
- Issue #2955[817] - Fix OS-Thread throttling
- PR #2953[818] - Making sure any hpx.os_threads=N supplied through a –hpx::config file is taken into account
- PR #2952[819] - Removing wrong call to cleanup_terminated_locked
- PR #2951[820] - Revert "Make sure the function vtables are initialized before use"
- PR #2950[821] - Fix a namespace compilation error when some schedulers are disabled
- Issue #2949[822] - master branch giving lockups on shutdown
- Issue #2947[823] - hpx.ini is not used correctly at initialization
- PR #2946[824] - Adding explicit feature test for thread_local
- PR #2945[825] - Make sure the function vtables are initialized before use
- PR #2944[826] - Attempting to solve affinity problems on CircleCI

[804] https://github.com/STEllAR-GROUP/hpx/pull/2969
[805] https://github.com/STEllAR-GROUP/hpx/pull/2968
[806] https://github.com/STEllAR-GROUP/hpx/pull/2967
[807] https://github.com/STEllAR-GROUP/hpx/pull/2966
[808] https://github.com/STEllAR-GROUP/hpx/pull/2965
[809] https://github.com/STEllAR-GROUP/hpx/issues/2964
[810] https://github.com/STEllAR-GROUP/hpx/pull/2963
[811] https://github.com/STEllAR-GROUP/hpx/pull/2962
[812] https://github.com/STEllAR-GROUP/hpx/pull/2961
[813] https://github.com/STEllAR-GROUP/hpx/issues/2959
[814] https://github.com/STEllAR-GROUP/hpx/issues/2958
[815] https://github.com/STEllAR-GROUP/hpx/pull/2957
[816] https://github.com/STEllAR-GROUP/hpx/issues/2956
[817] https://github.com/STEllAR-GROUP/hpx/issues/2955
[818] https://github.com/STEllAR-GROUP/hpx/pull/2953
[819] https://github.com/STEllAR-GROUP/hpx/pull/2952
[820] https://github.com/STEllAR-GROUP/hpx/pull/2951
[821] https://github.com/STEllAR-GROUP/hpx/pull/2950
[822] https://github.com/STEllAR-GROUP/hpx/issues/2949
[823] https://github.com/STEllAR-GROUP/hpx/issues/2947
[824] https://github.com/STEllAR-GROUP/hpx/pull/2946
[825] https://github.com/STEllAR-GROUP/hpx/pull/2945
[826] https://github.com/STEllAR-GROUP/hpx/pull/2944

- PR #2943[827] - Changing channel actions to be direct
- PR #2942[828] - Adding split_future for std::vector
- PR #2941[829] - Add a feature test to test for CXX11 override
- Issue #2940[830] - Add split_future for future<vector<T>>
- PR #2939[831] - Making error reporting during problems with setting affinity masks more verbose
- PR #2938[832] - Fix this various executors
- PR #2937[833] - Fix some typos in documentation
- PR #2934[834] - Remove the need for "complete" SFINAE checks
- PR #2933[835] - Making sure parallel::for_loop is executed in parallel if requested
- PR #2932[836] - Classify chunk_size_iterator to input iterator tag. (Fix #2866)
- Issue #2931[837] - –hpx:help triggers unusual error with clang build
- PR #2930[838] - Add #include files needed to set _POSIX_VERSION for debug check
- PR #2929[839] - Fix a couple of deprecated c++ features
- PR #2928[840] - Fixing execution parameters
- Issue #2927[841] - CMake warning: … cycle in constraint graph
- PR #2926[842] - Default pool rename
- Issue #2925[843] - Default pool cannot be renamed
- Issue #2924[844] - hpx:attach-debugger=startup does not work any more
- PR #2923[845] - Alloc membind
- PR #2922[846] - This fixes CircleCI errors when running with –hpx:bind=none
- PR #2921[847] - Custom pool executor was missing priority and stacksize options
- PR #2920[848] - Adding test to trigger problem reported in #2916
- PR #2919[849] - Make sure the resource_partitioner is properly destructed on hpx::finalize

[827] https://github.com/STEllAR-GROUP/hpx/pull/2943
[828] https://github.com/STEllAR-GROUP/hpx/pull/2942
[829] https://github.com/STEllAR-GROUP/hpx/pull/2941
[830] https://github.com/STEllAR-GROUP/hpx/issues/2940
[831] https://github.com/STEllAR-GROUP/hpx/pull/2939
[832] https://github.com/STEllAR-GROUP/hpx/pull/2938
[833] https://github.com/STEllAR-GROUP/hpx/pull/2937
[834] https://github.com/STEllAR-GROUP/hpx/pull/2934
[835] https://github.com/STEllAR-GROUP/hpx/pull/2933
[836] https://github.com/STEllAR-GROUP/hpx/pull/2932
[837] https://github.com/STEllAR-GROUP/hpx/issues/2931
[838] https://github.com/STEllAR-GROUP/hpx/pull/2930
[839] https://github.com/STEllAR-GROUP/hpx/pull/2929
[840] https://github.com/STEllAR-GROUP/hpx/pull/2928
[841] https://github.com/STEllAR-GROUP/hpx/issues/2927
[842] https://github.com/STEllAR-GROUP/hpx/pull/2926
[843] https://github.com/STEllAR-GROUP/hpx/issues/2925
[844] https://github.com/STEllAR-GROUP/hpx/issues/2924
[845] https://github.com/STEllAR-GROUP/hpx/pull/2923
[846] https://github.com/STEllAR-GROUP/hpx/pull/2922
[847] https://github.com/STEllAR-GROUP/hpx/pull/2921
[848] https://github.com/STEllAR-GROUP/hpx/pull/2920
[849] https://github.com/STEllAR-GROUP/hpx/pull/2919

- Issue #2918[850] - hpx::init calls wrong (first) callback when called multiple times
- PR #2917[851] - Adding util::checkpoint
- Issue #2916[852] - Weird runtime failures when using a channel and chained continuations
- PR #2915[853] - Introduce executor parameters customization points
- Issue #2914[854] - Task assignment to current Pool has unintended consequences
- PR #2913[855] - Fix rp hang
- PR #2912[856] - Update contributors
- PR #2911[857] - Fixing CUDA problems
- PR #2910[858] - Improve error reporting for process component on POSIX systems
- PR #2909[859] - Fix typo in include path
- PR #2908[860] - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2907[861] - Optionaly force-delete remaining channel items on close
- PR #2906[862] - Making sure generated performance counter names are correct
- Issue #2905[863] - collecting idle-rate performance counters on multiple localities produces an error
- Issue #2904[864] - build broken for Intel 17 compilers
- PR #2903[865] - Documentation Updates– Adding New People
- PR #2902[866] - Fixing service_executor
- PR #2901[867] - Fixing partitioned_vector creation
- PR #2900[868] - Add numa-balanced mode to hpx::bind, spread cores over numa domains
- Issue #2899[869] - hpx::bind does not have a mode that balances cores over numa domains
- PR #2898[870] - Adding missing #include and missing guard for optional code section
- PR #2897[871] - Removing dependency on Boost.ICL
- Issue #2896[872] - Debug build fails without -fpermissive with GCC 7.1 and Boost 1.65

[850] https://github.com/STEllAR-GROUP/hpx/issues/2918
[851] https://github.com/STEllAR-GROUP/hpx/pull/2917
[852] https://github.com/STEllAR-GROUP/hpx/issues/2916
[853] https://github.com/STEllAR-GROUP/hpx/pull/2915
[854] https://github.com/STEllAR-GROUP/hpx/issues/2914
[855] https://github.com/STEllAR-GROUP/hpx/pull/2913
[856] https://github.com/STEllAR-GROUP/hpx/pull/2912
[857] https://github.com/STEllAR-GROUP/hpx/pull/2911
[858] https://github.com/STEllAR-GROUP/hpx/pull/2910
[859] https://github.com/STEllAR-GROUP/hpx/pull/2909
[860] https://github.com/STEllAR-GROUP/hpx/pull/2908
[861] https://github.com/STEllAR-GROUP/hpx/pull/2907
[862] https://github.com/STEllAR-GROUP/hpx/pull/2906
[863] https://github.com/STEllAR-GROUP/hpx/issues/2905
[864] https://github.com/STEllAR-GROUP/hpx/issues/2904
[865] https://github.com/STEllAR-GROUP/hpx/pull/2903
[866] https://github.com/STEllAR-GROUP/hpx/pull/2902
[867] https://github.com/STEllAR-GROUP/hpx/pull/2901
[868] https://github.com/STEllAR-GROUP/hpx/pull/2900
[869] https://github.com/STEllAR-GROUP/hpx/issues/2899
[870] https://github.com/STEllAR-GROUP/hpx/pull/2898
[871] https://github.com/STEllAR-GROUP/hpx/pull/2897
[872] https://github.com/STEllAR-GROUP/hpx/issues/2896

- PR #2895[873] - Fixing SLURM environment parsing
- PR #2894[874] - Fix incorrect handling of compile definition with value 0
- Issue #2893[875] - Disabling schedulers causes build errors
- PR #2892[876] - added list serializer
- PR #2891[877] - Resource Partitioner Fixes
- Issue #2890[878] - Destroying a non-empty channel causes an assertion failure
- PR #2889[879] - Add check for libatomic
- PR #2888[880] - Fix compilation problems if HPX_WITH_ITT_NOTIFY=ON
- PR #2887[881] - Adapt broadcast() to non-unwrapping async<Action>
- PR #2886[882] - Replace Boost.Random with C++11 <random>
- Issue #2885[883] - regression in broadcast?
- Issue #2884[884] - linking `-latomic` is not portable
- PR #2883[885] - Explicitly set -pthread flag if available
- PR #2882[886] - Wrap boost::format uses
- Issue #2881[887] - hpx not compiling with `HPX_WITH_ITTNOTIFY=On`
- Issue #2880[888] - hpx::bind scatter/balanced give wrong pu masks
- PR #2878[889] - Fix incorrect pool usage masks setup in RP/thread manager
- PR #2877[890] - Require `std::array` by default
- PR #2875[891] - Deprecate use of BOOST_ASSERT
- PR #2874[892] - Changed serialization of boost.variant to use variadic templates
- Issue #2873[893] - building with parcelport_mpi fails on cori
- PR #2871[894] - Adding missing support for throttling scheduler
- PR #2870[895] - Disambiguate use of base_lco_with_value macros with channel

[873] https://github.com/STEllAR-GROUP/hpx/pull/2895
[874] https://github.com/STEllAR-GROUP/hpx/pull/2894
[875] https://github.com/STEllAR-GROUP/hpx/issues/2893
[876] https://github.com/STEllAR-GROUP/hpx/pull/2892
[877] https://github.com/STEllAR-GROUP/hpx/pull/2891
[878] https://github.com/STEllAR-GROUP/hpx/issues/2890
[879] https://github.com/STEllAR-GROUP/hpx/pull/2889
[880] https://github.com/STEllAR-GROUP/hpx/pull/2888
[881] https://github.com/STEllAR-GROUP/hpx/pull/2887
[882] https://github.com/STEllAR-GROUP/hpx/pull/2886
[883] https://github.com/STEllAR-GROUP/hpx/issues/2885
[884] https://github.com/STEllAR-GROUP/hpx/issues/2884
[885] https://github.com/STEllAR-GROUP/hpx/pull/2883
[886] https://github.com/STEllAR-GROUP/hpx/pull/2882
[887] https://github.com/STEllAR-GROUP/hpx/issues/2881
[888] https://github.com/STEllAR-GROUP/hpx/issues/2880
[889] https://github.com/STEllAR-GROUP/hpx/pull/2878
[890] https://github.com/STEllAR-GROUP/hpx/pull/2877
[891] https://github.com/STEllAR-GROUP/hpx/pull/2875
[892] https://github.com/STEllAR-GROUP/hpx/pull/2874
[893] https://github.com/STEllAR-GROUP/hpx/issues/2873
[894] https://github.com/STEllAR-GROUP/hpx/pull/2871
[895] https://github.com/STEllAR-GROUP/hpx/pull/2870

- Issue #2869[896] - Difficulty compiling `HPX_REGISTER_CHANNEL_DECLARATION(double)`
- PR #2868[897] - Removing uneeded assert
- PR #2867[898] - Implement parallel::unique
- Issue #2866[899] - The chunk_size_iterator violates multipass guarantee
- PR #2865[900] - Only use sched_getcpu on linux machines
- PR #2864[901] - Create redistribution archive for successful builds
- PR #2863[902] - Replace casts/assignments with hard-coded memcpy operations
- Issue #2862[903] - sched_getcpu not available on MacOS
- PR #2861[904] - Fixing unmatched header defines and recursive inclusion of threadmanager
- Issue #2860[905] - Master program fails with assertion 'type == data_type_address' failed: HPX(assertion_failure)
- Issue #2852[906] - Support for ARM64
- PR #2858[907] - Fix misplaced #if #endif's that cause build failure without THREAD_CUMULATIVE_COUNTS
- PR #2857[908] - Fix some listing in documentation
- PR #2856[909] - Fixing component handling for lcos
- PR #2855[910] - Add documentation for coarrays
- PR #2854[911] - Support ARM64 in timestamps
- PR #2853[912] - Update Table 17. Non-modifying Parallel Algorithms in Documentation
- PR #2851[913] - Allowing for non-default-constructible component types
- PR #2850[914] - Enable returning future<R> from actions where R is not default-constructible
- PR #2849[915] - Unify serialization of non-default-constructable types
- Issue #2848[916] - Components have to be default constructible
- Issue #2847[917] - Returning a future<R> where R is not default-constructable broken
- Issue #2846[918] - Unify serialization of non-default-constructible types

[896] https://github.com/STEllAR-GROUP/hpx/issues/2869
[897] https://github.com/STEllAR-GROUP/hpx/pull/2868
[898] https://github.com/STEllAR-GROUP/hpx/pull/2867
[899] https://github.com/STEllAR-GROUP/hpx/issues/2866
[900] https://github.com/STEllAR-GROUP/hpx/pull/2865
[901] https://github.com/STEllAR-GROUP/hpx/pull/2864
[902] https://github.com/STEllAR-GROUP/hpx/pull/2863
[903] https://github.com/STEllAR-GROUP/hpx/issues/2862
[904] https://github.com/STEllAR-GROUP/hpx/pull/2861
[905] https://github.com/STEllAR-GROUP/hpx/issues/2860
[906] https://github.com/STEllAR-GROUP/hpx/issues/2852
[907] https://github.com/STEllAR-GROUP/hpx/pull/2858
[908] https://github.com/STEllAR-GROUP/hpx/pull/2857
[909] https://github.com/STEllAR-GROUP/hpx/pull/2856
[910] https://github.com/STEllAR-GROUP/hpx/pull/2855
[911] https://github.com/STEllAR-GROUP/hpx/pull/2854
[912] https://github.com/STEllAR-GROUP/hpx/pull/2853
[913] https://github.com/STEllAR-GROUP/hpx/pull/2851
[914] https://github.com/STEllAR-GROUP/hpx/pull/2850
[915] https://github.com/STEllAR-GROUP/hpx/pull/2849
[916] https://github.com/STEllAR-GROUP/hpx/issues/2848
[917] https://github.com/STEllAR-GROUP/hpx/issues/2847
[918] https://github.com/STEllAR-GROUP/hpx/issues/2846

- PR #2845[919] - Add Visual Studio 2015 to the tested toolchains in Appveyor
- Issue #2844[920] - Change the appveyor build to use the minimal required MSVC version
- Issue #2843[921] - multi node hello_world hangs
- PR #2842[922] - Correcting Spelling mistake in docs
- PR #2841[923] - Fix usage of std::aligned_storage
- PR #2840[924] - Remove constexpr from a void function
- Issue #2839[925] - memcpy buffer overflow: load_construct_data() and std::complex members
- Issue #2835[926] - `constexpr` functions with `void` return type break compilation with CUDA 8.0
- Issue #2834[927] - One suspicion in parallel::detail::handle_exception
- PR #2833[928] - Implement parallel::merge
- PR #2832[929] - Fix a strange thing in parallel::util::detail::handle_local_exceptions. (Fix #2818)
- PR #2830[930] - Break the debugger when a test failed
- Issue #2831[931] - `parallel/executors/execution_fwd.hpp` causes compilation failure in C++11 mode.
- PR #2829[932] - Implement an API for asynchronous pack traversal
- PR #2828[933] - Split unit test builds on CircleCI to avoid timeouts
- Issue #2827[934] - failure to compile hello_world example with -Werror
- PR #2824[935] - Making sure promises are marked as started when used as continuations
- PR #2823[936] - Add documentation for partitioned_vector_view
- Issue #2822[937] - Yet another issue with wait_for similar to #2796
- PR #2821[938] - Fix bugs and improve that about HPX_HAVE_CXX11_AUTO_RETURN_VALUE of CMake
- PR #2820[939] - Support C++11 in benchmark codes of parallel::partition and parallel::partition_copy
- PR #2819[940] - Fix compile errors in unit test of container version of parallel::partition
- Issue #2818[941] - A strange thing in parallel::util::detail::handle_local_exceptions

919 https://github.com/STEllAR-GROUP/hpx/pull/2845
920 https://github.com/STEllAR-GROUP/hpx/issues/2844
921 https://github.com/STEllAR-GROUP/hpx/issues/2843
922 https://github.com/STEllAR-GROUP/hpx/pull/2842
923 https://github.com/STEllAR-GROUP/hpx/pull/2841
924 https://github.com/STEllAR-GROUP/hpx/pull/2840
925 https://github.com/STEllAR-GROUP/hpx/issues/2839
926 https://github.com/STEllAR-GROUP/hpx/issues/2835
927 https://github.com/STEllAR-GROUP/hpx/issues/2834
928 https://github.com/STEllAR-GROUP/hpx/pull/2833
929 https://github.com/STEllAR-GROUP/hpx/pull/2832
930 https://github.com/STEllAR-GROUP/hpx/pull/2830
931 https://github.com/STEllAR-GROUP/hpx/issues/2831
932 https://github.com/STEllAR-GROUP/hpx/pull/2829
933 https://github.com/STEllAR-GROUP/hpx/pull/2828
934 https://github.com/STEllAR-GROUP/hpx/issues/2827
935 https://github.com/STEllAR-GROUP/hpx/pull/2824
936 https://github.com/STEllAR-GROUP/hpx/pull/2823
937 https://github.com/STEllAR-GROUP/hpx/issues/2822
938 https://github.com/STEllAR-GROUP/hpx/pull/2821
939 https://github.com/STEllAR-GROUP/hpx/pull/2820
940 https://github.com/STEllAR-GROUP/hpx/pull/2819
941 https://github.com/STEllAR-GROUP/hpx/issues/2818

- Issue #2815[942] - HPX fails to compile with HPX_WITH_CUDA=ON and the new CUDA 9.0 RC
- Issue #2814[943] - Using 'gmakeN' after 'cmake' produces error in src/CMakeFiles/hpx.dir/runtime/agas/addressing_service.cpp.o
- PR #2813[944] - Properly support [[noreturn]] attribute if available
- Issue #2812[945] - Compilation fails with gcc 7.1.1
- PR #2811[946] - Adding hpx::launch::lazy and support for async, dataflow, and future::then
- PR #2810[947] - Add option allowing to disable deprecation warning
- PR #2809[948] - Disable throttling scheduler if HWLOC is not found/used
- PR #2808[949] - Fix compile errors on some environments of parallel::partition
- Issue #2807[950] - Difficulty building with `HPX_WITH_HWLOC=Off`
- PR #2806[951] - Partitioned vector
- PR #2805[952] - Serializing collections with non-default constructible data
- PR #2802[953] - Fix FreeBSD 11
- Issue #2801[954] - Rate limiting techniques in io_service
- Issue #2800[955] - New Launch Policy: async_if
- PR #2799[956] - Fix a unit test failure on GCC in tuple_cat
- PR #2798[957] - bump minimum required cmake to 3.0 in test
- PR #2797[958] - Making sure future::wait_for et.al. work properly for action results
- Issue #2796[959] - wait_for does always in "deferred" state for calls on remote localities
- Issue #2795[960] - Serialization of types without default constructor
- PR #2794[961] - Fixing test for partitioned_vector iteration
- PR #2792[962] - Implemented segmented find and its variations for partitioned vector
- PR #2791[963] - Circumvent scary warning about placement new
- PR #2790[964] - Fix OSX build

942 https://github.com/STEllAR-GROUP/hpx/issues/2815
943 https://github.com/STEllAR-GROUP/hpx/issues/2814
944 https://github.com/STEllAR-GROUP/hpx/pull/2813
945 https://github.com/STEllAR-GROUP/hpx/issues/2812
946 https://github.com/STEllAR-GROUP/hpx/pull/2811
947 https://github.com/STEllAR-GROUP/hpx/pull/2810
948 https://github.com/STEllAR-GROUP/hpx/pull/2809
949 https://github.com/STEllAR-GROUP/hpx/pull/2808
950 https://github.com/STEllAR-GROUP/hpx/issues/2807
951 https://github.com/STEllAR-GROUP/hpx/pull/2806
952 https://github.com/STEllAR-GROUP/hpx/pull/2805
953 https://github.com/STEllAR-GROUP/hpx/pull/2802
954 https://github.com/STEllAR-GROUP/hpx/issues/2801
955 https://github.com/STEllAR-GROUP/hpx/issues/2800
956 https://github.com/STEllAR-GROUP/hpx/pull/2799
957 https://github.com/STEllAR-GROUP/hpx/pull/2798
958 https://github.com/STEllAR-GROUP/hpx/pull/2797
959 https://github.com/STEllAR-GROUP/hpx/issues/2796
960 https://github.com/STEllAR-GROUP/hpx/issues/2795
961 https://github.com/STEllAR-GROUP/hpx/pull/2794
962 https://github.com/STEllAR-GROUP/hpx/pull/2792
963 https://github.com/STEllAR-GROUP/hpx/pull/2791
964 https://github.com/STEllAR-GROUP/hpx/pull/2790

- PR #2789[965] - Resource partitioner
- PR #2788[966] - Adapt parallel::is_heap and parallel::is_heap_until to Ranges TS
- PR #2787[967] - Unwrap hotfixes
- PR #2786[968] - Update CMake Minimum Version to 3.3.2 (refs #2565)
- Issue #2785[969] - Issues with masks and cpuset
- PR #2784[970] - Error with reduce and transform reduce fixed
- PR #2783[971] - StackOverflow integration with libsigsegv
- PR #2782[972] - Replace boost::atomic with std::atomic (where possible)
- PR #2781[973] - Check for and optionally use [[deprecated]] attribute
- PR #2780[974] - Adding empty (but non-trivial) destructor to circumvent warnings
- PR #2779[975] - Exception info tweaks
- PR #2778[976] - Implement parallel::partition
- PR #2777[977] - Improve error handling in gather_here/gather_there
- PR #2776[978] - Fix a bug in compiler version check
- PR #2775[979] - Fix compilation when HPX_WITH_LOGGING is OFF
- PR #2774[980] - Removing dependency on Boost.Date_Time
- PR #2773[981] - Add sync_images() method to spmd_block class
- PR #2772[982] - Adding documentation for PAPI counters
- PR #2771[983] - Removing boost preprocessor dependency
- PR #2770[984] - Adding test, fixing deadlock in config registry
- PR #2769[985] - Remove some other warnings and errors detected by clang 5.0
- Issue #2768[986] - Is there iterator tag for HPX?
- PR #2767[987] - Improvements to continuation annotation

[965] https://github.com/STEllAR-GROUP/hpx/pull/2789
[966] https://github.com/STEllAR-GROUP/hpx/pull/2788
[967] https://github.com/STEllAR-GROUP/hpx/pull/2787
[968] https://github.com/STEllAR-GROUP/hpx/pull/2786
[969] https://github.com/STEllAR-GROUP/hpx/issues/2785
[970] https://github.com/STEllAR-GROUP/hpx/pull/2784
[971] https://github.com/STEllAR-GROUP/hpx/pull/2783
[972] https://github.com/STEllAR-GROUP/hpx/pull/2782
[973] https://github.com/STEllAR-GROUP/hpx/pull/2781
[974] https://github.com/STEllAR-GROUP/hpx/pull/2780
[975] https://github.com/STEllAR-GROUP/hpx/pull/2779
[976] https://github.com/STEllAR-GROUP/hpx/pull/2778
[977] https://github.com/STEllAR-GROUP/hpx/pull/2777
[978] https://github.com/STEllAR-GROUP/hpx/pull/2776
[979] https://github.com/STEllAR-GROUP/hpx/pull/2775
[980] https://github.com/STEllAR-GROUP/hpx/pull/2774
[981] https://github.com/STEllAR-GROUP/hpx/pull/2773
[982] https://github.com/STEllAR-GROUP/hpx/pull/2772
[983] https://github.com/STEllAR-GROUP/hpx/pull/2771
[984] https://github.com/STEllAR-GROUP/hpx/pull/2770
[985] https://github.com/STEllAR-GROUP/hpx/pull/2769
[986] https://github.com/STEllAR-GROUP/hpx/issues/2768
[987] https://github.com/STEllAR-GROUP/hpx/pull/2767

- PR #2765[988] - gcc split stack support for HPX threads #620
- PR #2764[989] - Fix some uses of begin/end, remove unnecessary includes
- PR #2763[990] - Bump minimal Boost version to 1.55.0
- PR #2762[991] - hpx::partitioned_vector serializer
- PR #2761[992] - Adding configuration summary to cmake output and –hpx:info
- PR #2760[993] - Removing 1d_hydro example as it is broken
- PR #2758[994] - Remove various warnings detected by clang 5.0
- Issue #2757[995] - In case of a “raw thread” is needed per core for implementing parallel algorithm, what is good practice in HPX?
- PR #2756[996] - Allowing for LCOs to be simple components
- PR #2755[997] - Removing make_index_pack_unrolled
- PR #2754[998] - Implement parallel::unique_copy
- PR #2753[999] - Fixing detection of [[fallthrough]] attribute
- PR #2752[1000] - New thread priority names
- PR #2751[1001] - Replace boost::exception with proposed exception_info
- PR #2750[1002] - Replace boost::iterator_range
- PR #2749[1003] - Fixing hdf5 examples
- Issue #2748[1004] - HPX fails to build with enabled hdf5 examples
- Issue #2747[1005] - Inherited task priorities break certain DAG optimizations
- Issue #2746[1006] - HPX segfaulting with valgrind
- PR #2745[1007] - Adding extended arithmetic performance counters
- PR #2744[1008] - Adding ability to statistics counters to reset base counter
- Issue #2743[1009] - Statistics counter does not support reseting
- PR #2742[1010] - Making sure Vc V2 builds without additional HPX configuration flags

988 https://github.com/STEllAR-GROUP/hpx/pull/2765
989 https://github.com/STEllAR-GROUP/hpx/pull/2764
990 https://github.com/STEllAR-GROUP/hpx/pull/2763
991 https://github.com/STEllAR-GROUP/hpx/pull/2762
992 https://github.com/STEllAR-GROUP/hpx/pull/2761
993 https://github.com/STEllAR-GROUP/hpx/pull/2760
994 https://github.com/STEllAR-GROUP/hpx/pull/2758
995 https://github.com/STEllAR-GROUP/hpx/issues/2757
996 https://github.com/STEllAR-GROUP/hpx/pull/2756
997 https://github.com/STEllAR-GROUP/hpx/pull/2755
998 https://github.com/STEllAR-GROUP/hpx/pull/2754
999 https://github.com/STEllAR-GROUP/hpx/pull/2753
1000 https://github.com/STEllAR-GROUP/hpx/pull/2752
1001 https://github.com/STEllAR-GROUP/hpx/pull/2751
1002 https://github.com/STEllAR-GROUP/hpx/pull/2750
1003 https://github.com/STEllAR-GROUP/hpx/pull/2749
1004 https://github.com/STEllAR-GROUP/hpx/issues/2748
1005 https://github.com/STEllAR-GROUP/hpx/issues/2747
1006 https://github.com/STEllAR-GROUP/hpx/issues/2746
1007 https://github.com/STEllAR-GROUP/hpx/pull/2745
1008 https://github.com/STEllAR-GROUP/hpx/pull/2744
1009 https://github.com/STEllAR-GROUP/hpx/issues/2743
1010 https://github.com/STEllAR-GROUP/hpx/pull/2742

- PR #2741[1011] - Deprecate unwrapped and implement unwrap and unwrapping
- PR #2740[1012] - Coroutine stackoverflow detection for linux/posix; Issue #2408
- PR #2739[1013] - Add files via upload
- PR #2738[1014] - Appveyor support
- PR #2737[1015] - Fixing 2735
- Issue #2736[1016] - 1d_hydro example does't work
- Issue #2735[1017] - partitioned_vector_subview test failing
- PR #2734[1018] - Add C++11 range utilities
- PR #2733[1019] - Adapting iterator requirements for parallel algorithms
- PR #2732[1020] - Integrate C++ Co-arrays
- PR #2731[1021] - Adding on_migrated event handler to migratable component instances
- Issue #2729[1022] - Add on_migrated() event handler to migratable components
- Issue #2728[1023] - Why Projection is needed in parallel algorithms?
- PR #2727[1024] - Cmake files for StackOverflow Detection
- PR #2726[1025] - CMake for Stack Overflow Detection
- PR #2725[1026] - Implemented segmented algorithms for partitioned vector
- PR #2724[1027] - Fix examples in Action documentation
- PR #2723[1028] - Enable lcos::channel<T>::register_as
- Issue #2722[1029] - channel register_as() failing on compilation
- PR #2721[1030] - Mind map
- PR #2720[1031] - reorder forward declarations to get rid of C++14-only auto return types
- PR #2719[1032] - Add documentation for partitioned_vector and add features in pack.hpp
- Issue #2718[1033] - Some forward declarations in execution_fwd.hpp aren't C++11-compatible

[1011] https://github.com/STEllAR-GROUP/hpx/pull/2741
[1012] https://github.com/STEllAR-GROUP/hpx/pull/2740
[1013] https://github.com/STEllAR-GROUP/hpx/pull/2739
[1014] https://github.com/STEllAR-GROUP/hpx/pull/2738
[1015] https://github.com/STEllAR-GROUP/hpx/pull/2737
[1016] https://github.com/STEllAR-GROUP/hpx/issues/2736
[1017] https://github.com/STEllAR-GROUP/hpx/issues/2735
[1018] https://github.com/STEllAR-GROUP/hpx/pull/2734
[1019] https://github.com/STEllAR-GROUP/hpx/pull/2733
[1020] https://github.com/STEllAR-GROUP/hpx/pull/2732
[1021] https://github.com/STEllAR-GROUP/hpx/pull/2731
[1022] https://github.com/STEllAR-GROUP/hpx/issues/2729
[1023] https://github.com/STEllAR-GROUP/hpx/issues/2728
[1024] https://github.com/STEllAR-GROUP/hpx/pull/2727
[1025] https://github.com/STEllAR-GROUP/hpx/pull/2726
[1026] https://github.com/STEllAR-GROUP/hpx/pull/2725
[1027] https://github.com/STEllAR-GROUP/hpx/pull/2724
[1028] https://github.com/STEllAR-GROUP/hpx/pull/2723
[1029] https://github.com/STEllAR-GROUP/hpx/issues/2722
[1030] https://github.com/STEllAR-GROUP/hpx/pull/2721
[1031] https://github.com/STEllAR-GROUP/hpx/pull/2720
[1032] https://github.com/STEllAR-GROUP/hpx/pull/2719
[1033] https://github.com/STEllAR-GROUP/hpx/issues/2718

- PR #2717[1034] - Config support for fallthrough attribute
- PR #2716[1035] - Implement parallel::partition_copy
- PR #2715[1036] - initial import of icu string serializer
- PR #2714[1037] - initial import of valarray serializer
- PR #2713[1038] - Remove slashes before CMAKE_FILES_DIRECTORY variables
- PR #2712[1039] - Fixing wait for 1751
- PR #2711[1040] - Adjust code for minimal supported GCC having being bumped to 4.9
- PR #2710[1041] - Adding code of conduct
- PR #2709[1042] - Fixing UB in destroy tests
- PR #2708[1043] - Add inline to prevent multiple definition issue
- Issue #2707[1044] - Multiple defined symbols for task_block.hpp in VS2015
- PR #2706[1045] - Adding .clang-format file
- PR #2704[1046] - Add a synchronous mapping API
- Issue #2703[1047] - Request: Add the .clang-format file to the repository
- Issue #2702[1048] - STEllAR-GROUP/Vc slower than VCv1 possibly due to wrong instructions generated
- Issue #2701[1049] - Datapar with STEllAR-GROUP/Vc requires obscure flag
- Issue #2700[1050] - Naming inconsistency in parallel algorithms
- Issue #2699[1051] - Iterator requirements are different from standard in parallel copy_if.
- PR #2698[1052] - Properly releasing parcelport write handlers
- Issue #2697[1053] - Compile error in addressing_service.cpp
- Issue #2696[1054] - Building and using HPX statically: undefined references from runtime_support_server.cpp
- Issue #2695[1055] - Executor changes cause compilation failures
- PR #2694[1056] - Refining C++ language mode detection for MSVC

[1034] https://github.com/STEllAR-GROUP/hpx/pull/2717
[1035] https://github.com/STEllAR-GROUP/hpx/pull/2716
[1036] https://github.com/STEllAR-GROUP/hpx/pull/2715
[1037] https://github.com/STEllAR-GROUP/hpx/pull/2714
[1038] https://github.com/STEllAR-GROUP/hpx/pull/2713
[1039] https://github.com/STEllAR-GROUP/hpx/pull/2712
[1040] https://github.com/STEllAR-GROUP/hpx/pull/2711
[1041] https://github.com/STEllAR-GROUP/hpx/pull/2710
[1042] https://github.com/STEllAR-GROUP/hpx/pull/2709
[1043] https://github.com/STEllAR-GROUP/hpx/pull/2708
[1044] https://github.com/STEllAR-GROUP/hpx/issues/2707
[1045] https://github.com/STEllAR-GROUP/hpx/pull/2706
[1046] https://github.com/STEllAR-GROUP/hpx/pull/2704
[1047] https://github.com/STEllAR-GROUP/hpx/issues/2703
[1048] https://github.com/STEllAR-GROUP/hpx/issues/2702
[1049] https://github.com/STEllAR-GROUP/hpx/issues/2701
[1050] https://github.com/STEllAR-GROUP/hpx/issues/2700
[1051] https://github.com/STEllAR-GROUP/hpx/issues/2699
[1052] https://github.com/STEllAR-GROUP/hpx/pull/2698
[1053] https://github.com/STEllAR-GROUP/hpx/issues/2697
[1054] https://github.com/STEllAR-GROUP/hpx/issues/2696
[1055] https://github.com/STEllAR-GROUP/hpx/issues/2695
[1056] https://github.com/STEllAR-GROUP/hpx/pull/2694

- PR #2693[1057] - P0443 r2
- PR #2692[1058] - Partially reverting changes to parcel_await
- Issue #2689[1059] - HPX build fails when HPX_WITH_CUDA is enabled
- PR #2688[1060] - Make Cuda Clang builds pass
- PR #2687[1061] - Add an is_tuple_like trait for sequenceable type detection
- PR #2686[1062] - Allowing throttling scheduler to be used without idle backoff
- PR #2685[1063] - Add support of std::array to hpx::util::tuple_size and tuple_element
- PR #2684[1064] - Adding new statistics performance counters
- PR #2683[1065] - Replace boost::exception_ptr with std::exception_ptr
- Issue #2682[1066] - HPX does not compile with HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF
- PR #2681[1067] - Attempt to fix problem in managed_component_base
- PR #2680[1068] - Fix bad size during archive creation
- Issue #2679[1069] - Mismatch between size of archive and container
- Issue #2678[1070] - In parallel algorithm, other tasks are executed to the end even if an exception occurs in any task.
- PR #2677[1071] - Adding include check for std::addressof
- PR #2676[1072] - Adding parallel::destroy and destroy_n
- PR #2675[1073] - Making sure statistics counters work as expected
- PR #2674[1074] - Turning assertions into exceptions
- PR #2673[1075] - Inhibit direct conversion from future<future<T>> –> future<void>
- PR #2672[1076] - C++17 invoke forms
- PR #2671[1077] - Adding uninitialized_value_construct and uninitialized_value_construct_n
- PR #2670[1078] - Integrate spmd multidimensionnal views for partitioned_vectors
- PR #2669[1079] - Adding uninitialized_default_construct and uninitialized_default_construct_n

[1057] https://github.com/STEllAR-GROUP/hpx/pull/2693
[1058] https://github.com/STEllAR-GROUP/hpx/pull/2692
[1059] https://github.com/STEllAR-GROUP/hpx/issues/2689
[1060] https://github.com/STEllAR-GROUP/hpx/pull/2688
[1061] https://github.com/STEllAR-GROUP/hpx/pull/2687
[1062] https://github.com/STEllAR-GROUP/hpx/pull/2686
[1063] https://github.com/STEllAR-GROUP/hpx/pull/2685
[1064] https://github.com/STEllAR-GROUP/hpx/pull/2684
[1065] https://github.com/STEllAR-GROUP/hpx/pull/2683
[1066] https://github.com/STEllAR-GROUP/hpx/issues/2682
[1067] https://github.com/STEllAR-GROUP/hpx/pull/2681
[1068] https://github.com/STEllAR-GROUP/hpx/pull/2680
[1069] https://github.com/STEllAR-GROUP/hpx/issues/2679
[1070] https://github.com/STEllAR-GROUP/hpx/issues/2678
[1071] https://github.com/STEllAR-GROUP/hpx/pull/2677
[1072] https://github.com/STEllAR-GROUP/hpx/pull/2676
[1073] https://github.com/STEllAR-GROUP/hpx/pull/2675
[1074] https://github.com/STEllAR-GROUP/hpx/pull/2674
[1075] https://github.com/STEllAR-GROUP/hpx/pull/2673
[1076] https://github.com/STEllAR-GROUP/hpx/pull/2672
[1077] https://github.com/STEllAR-GROUP/hpx/pull/2671
[1078] https://github.com/STEllAR-GROUP/hpx/pull/2670
[1079] https://github.com/STEllAR-GROUP/hpx/pull/2669

- PR #2668[1080] - Fixing documentation index
- Issue #2667[1081] - Ambiguity of nested hpx::future<void>'s
- Issue #2666[1082] - Statistics Performance counter is not working
- PR #2664[1083] - Adding uninitialized_move and uninitialized_move_n
- Issue #2663[1084] - Seg fault in managed_component::get_base_gid, possibly cause by util::reinitializable_static
- Issue #2662[1085] - Crash in managed_component::get_base_gid due to problem with util::reinitializable_static
- PR #2665[1086] - Hide the `detail` namespace in doxygen per default
- PR #2660[1087] - Add documentation to hpx::util::unwrapped and hpx::util::unwrapped2
- PR #2659[1088] - Improve integration with vcpkg
- PR #2658[1089] - Unify access_data trait for use in both, serialization and de-serialization
- PR #2657[1090] - Removing hpx::lcos::queue<T>
- PR #2656[1091] - Reduce MAX_TERMINATED_THREADS default, improve memory use on manycore cpus
- PR #2655[1092] - Mainteinance for emulate-deleted macros
- PR #2654[1093] - Implement parallel is_heap and is_heap_until
- PR #2653[1094] - Drop support for VS2013
- PR #2652[1095] - This patch makes sure that all parcels in a batch are properly handled
- PR #2649[1096] - Update docs (Table 18) - move transform to end
- Issue #2647[1097] - hpx::parcelset::detail::parcel_data::**has_continuation_** is unitialized
- Issue #2644[1098] - Some .vcxproj in the HPX.sln fail to build
- Issue #2641[1099] - `hpx::lcos::queue` should be deprecated
- PR #2640[1100] - A new throttling policy with public APIs to suspend/resume
- PR #2639[1101] - Fix a tiny typo in tutorial.
- Issue #2638[1102] - Invalid return type 'void' of constexpr function

[1080] https://github.com/STEllAR-GROUP/hpx/pull/2668
[1081] https://github.com/STEllAR-GROUP/hpx/issues/2667
[1082] https://github.com/STEllAR-GROUP/hpx/issues/2666
[1083] https://github.com/STEllAR-GROUP/hpx/pull/2664
[1084] https://github.com/STEllAR-GROUP/hpx/issues/2663
[1085] https://github.com/STEllAR-GROUP/hpx/issues/2662
[1086] https://github.com/STEllAR-GROUP/hpx/pull/2665
[1087] https://github.com/STEllAR-GROUP/hpx/pull/2660
[1088] https://github.com/STEllAR-GROUP/hpx/pull/2659
[1089] https://github.com/STEllAR-GROUP/hpx/pull/2658
[1090] https://github.com/STEllAR-GROUP/hpx/pull/2657
[1091] https://github.com/STEllAR-GROUP/hpx/pull/2656
[1092] https://github.com/STEllAR-GROUP/hpx/pull/2655
[1093] https://github.com/STEllAR-GROUP/hpx/pull/2654
[1094] https://github.com/STEllAR-GROUP/hpx/pull/2653
[1095] https://github.com/STEllAR-GROUP/hpx/pull/2652
[1096] https://github.com/STEllAR-GROUP/hpx/pull/2649
[1097] https://github.com/STEllAR-GROUP/hpx/issues/2647
[1098] https://github.com/STEllAR-GROUP/hpx/issues/2644
[1099] https://github.com/STEllAR-GROUP/hpx/issues/2641
[1100] https://github.com/STEllAR-GROUP/hpx/pull/2640
[1101] https://github.com/STEllAR-GROUP/hpx/pull/2639
[1102] https://github.com/STEllAR-GROUP/hpx/issues/2638

- PR #2636[1103] - Add and use HPX_MSVC_WARNING_PRAGMA for #pragma warning
- PR #2633[1104] - Distributed define_spmd_block
- PR #2632[1105] - Making sure container serialization uses size-compatible types
- PR #2631[1106] - Add lcos::local::one_element_channel
- PR #2629[1107] - Move unordered_map out of parcelport into hpx/concurrent
- PR #2628[1108] - Making sure that shutdown does not hang
- PR #2627[1109] - Fix serialization
- PR #2626[1110] - Generate `cmake_variables.qbk` and `cmake_toolchains.qbk` outside of the source tree
- PR #2625[1111] - Supporting -std=c++17 flag
- PR #2624[1112] - Fixing a small cmake typo
- PR #2622[1113] - Update CMake minimum required version to 3.0.2 (closes #2621)
- Issue #2621[1114] - Compiling hpx master fails with /usr/bin/ld: final link failed: Bad value
- PR #2620[1115] - Remove warnings due to some captured variables
- PR #2619[1116] - LF multiple parcels
- PR #2618[1117] - Some fixes to libfabric that didn't get caught before the merge
- PR #2617[1118] - Adding `hpx::local_new`
- PR #2616[1119] - Documentation: Extract all entities in order to autolink functions correctly
- Issue #2615[1120] - Documentation: Linking functions is broken
- PR #2614[1121] - Adding serialization for std::deque
- PR #2613[1122] - We need to link with boost.thread and boost.chrono if we use boost.context
- PR #2612[1123] - Making sure for_loop_n(par, ...) is actually executed in parallel
- PR #2611[1124] - Add documentation to invoke_fused and friends NFC
- PR #2610[1125] - Added reduction templates using an identity value

[1103] https://github.com/STEllAR-GROUP/hpx/pull/2636
[1104] https://github.com/STEllAR-GROUP/hpx/pull/2633
[1105] https://github.com/STEllAR-GROUP/hpx/pull/2632
[1106] https://github.com/STEllAR-GROUP/hpx/pull/2631
[1107] https://github.com/STEllAR-GROUP/hpx/pull/2629
[1108] https://github.com/STEllAR-GROUP/hpx/pull/2628
[1109] https://github.com/STEllAR-GROUP/hpx/pull/2627
[1110] https://github.com/STEllAR-GROUP/hpx/pull/2626
[1111] https://github.com/STEllAR-GROUP/hpx/pull/2625
[1112] https://github.com/STEllAR-GROUP/hpx/pull/2624
[1113] https://github.com/STEllAR-GROUP/hpx/pull/2622
[1114] https://github.com/STEllAR-GROUP/hpx/issues/2621
[1115] https://github.com/STEllAR-GROUP/hpx/pull/2620
[1116] https://github.com/STEllAR-GROUP/hpx/pull/2619
[1117] https://github.com/STEllAR-GROUP/hpx/pull/2618
[1118] https://github.com/STEllAR-GROUP/hpx/pull/2617
[1119] https://github.com/STEllAR-GROUP/hpx/pull/2616
[1120] https://github.com/STEllAR-GROUP/hpx/issues/2615
[1121] https://github.com/STEllAR-GROUP/hpx/pull/2614
[1122] https://github.com/STEllAR-GROUP/hpx/pull/2613
[1123] https://github.com/STEllAR-GROUP/hpx/pull/2612
[1124] https://github.com/STEllAR-GROUP/hpx/pull/2611
[1125] https://github.com/STEllAR-GROUP/hpx/pull/2610

- PR #2608[1126] - Fixing some unused vars in inspect
- PR #2607[1127] - Fixed build for mingw
- PR #2606[1128] - Supporting generic context for boost >= 1.61
- PR #2605[1129] - Parcelport libfabric3
- PR #2604[1130] - Adding allocator support to promise and friends
- PR #2603[1131] - Barrier hang
- PR #2602[1132] - Changes to scheduler to steal from one high-priority queue
- Issue #2601[1133] - High priority tasks are not executed first
- PR #2600[1134] - Compat fixes
- PR #2599[1135] - Compatibility layer for threading support
- PR #2598[1136] - V1.1
- PR #2597[1137] - Release V1.0
- PR #2592[1138] - First attempt to introduce spmd_block in hpx
- PR #2586[1139] - local_segment in segmented_iterator_traits
- Issue #2584[1140] - Add allocator support to promise, packaged_task and friends
- PR #2576[1141] - Add missing dependencies of cuda based tests
- PR #2575[1142] - Remove warnings due to some captured variables
- Issue #2574[1143] - MSVC 2015 Compiler crash when building HPX
- Issue #2568[1144] - Remove throttle_scheduler as it has been abandoned
- Issue #2566[1145] - Add an inline versioning namespace before 1.0 release
- Issue #2565[1146] - Raise minimal cmake version requirement
- PR #2556[1147] - Fixing scan partitioner
- PR #2546[1148] - Broadcast async

[1126] https://github.com/STEllAR-GROUP/hpx/pull/2608
[1127] https://github.com/STEllAR-GROUP/hpx/pull/2607
[1128] https://github.com/STEllAR-GROUP/hpx/pull/2606
[1129] https://github.com/STEllAR-GROUP/hpx/pull/2605
[1130] https://github.com/STEllAR-GROUP/hpx/pull/2604
[1131] https://github.com/STEllAR-GROUP/hpx/pull/2603
[1132] https://github.com/STEllAR-GROUP/hpx/pull/2602
[1133] https://github.com/STEllAR-GROUP/hpx/issues/2601
[1134] https://github.com/STEllAR-GROUP/hpx/pull/2600
[1135] https://github.com/STEllAR-GROUP/hpx/pull/2599
[1136] https://github.com/STEllAR-GROUP/hpx/pull/2598
[1137] https://github.com/STEllAR-GROUP/hpx/pull/2597
[1138] https://github.com/STEllAR-GROUP/hpx/pull/2592
[1139] https://github.com/STEllAR-GROUP/hpx/pull/2586
[1140] https://github.com/STEllAR-GROUP/hpx/issues/2584
[1141] https://github.com/STEllAR-GROUP/hpx/pull/2576
[1142] https://github.com/STEllAR-GROUP/hpx/pull/2575
[1143] https://github.com/STEllAR-GROUP/hpx/issues/2574
[1144] https://github.com/STEllAR-GROUP/hpx/issues/2568
[1145] https://github.com/STEllAR-GROUP/hpx/issues/2566
[1146] https://github.com/STEllAR-GROUP/hpx/issues/2565
[1147] https://github.com/STEllAR-GROUP/hpx/pull/2556
[1148] https://github.com/STEllAR-GROUP/hpx/pull/2546

- Issue #2543[1149] - make install fails due to a non-existing .so file
- PR #2495[1150] - wait_or_add_new returning thread_id_type
- Issue #2480[1151] - Unable to register new performance counter
- Issue #2471[1152] - no type named 'fcontext_t' in namespace
- Issue #2456[1153] - Re-implement hpx::util::unwrapped
- Issue #2455[1154] - Add more arithmetic performance counters
- PR #2454[1155] - Fix a couple of warnings and compiler errors
- PR #2453[1156] - Timed executor support
- PR #2447[1157] - Implementing new executor API (P0443)
- Issue #2439[1158] - Implement executor proposal
- Issue #2408[1159] - Stackoverflow detection for linux, e.g. based on libsigsegv
- PR #2377[1160] - Add a customization point for put_parcel so we can override actions
- Issue #2368[1161] - HPX_ASSERT problem
- Issue #2324[1162] - Change default number of threads used to the maximum of the system
- Issue #2266[1163] - hpx_0.9.99 make tests fail
- PR #2195[1164] - Support for code completion in VIM
- Issue #2137[1165] - Hpx does not compile over osx
- Issue #2092[1166] - make tests should just build the tests
- Issue #2026[1167] - Build HPX with Apple's clang
- Issue #1932[1168] - hpx with PBS fails on multiple localities
- PR #1914[1169] - Parallel heap algorithm implementations WIP
- Issue #1598[1170] - Disconnecting a locality results in segfault using heartbeat example
- Issue #1404[1171] - unwrapped doesn't work with movable only types

[1149] https://github.com/STEllAR-GROUP/hpx/issues/2543
[1150] https://github.com/STEllAR-GROUP/hpx/pull/2495
[1151] https://github.com/STEllAR-GROUP/hpx/issues/2480
[1152] https://github.com/STEllAR-GROUP/hpx/issues/2471
[1153] https://github.com/STEllAR-GROUP/hpx/issues/2456
[1154] https://github.com/STEllAR-GROUP/hpx/issues/2455
[1155] https://github.com/STEllAR-GROUP/hpx/pull/2454
[1156] https://github.com/STEllAR-GROUP/hpx/pull/2453
[1157] https://github.com/STEllAR-GROUP/hpx/pull/2447
[1158] https://github.com/STEllAR-GROUP/hpx/issues/2439
[1159] https://github.com/STEllAR-GROUP/hpx/issues/2408
[1160] https://github.com/STEllAR-GROUP/hpx/pull/2377
[1161] https://github.com/STEllAR-GROUP/hpx/issues/2368
[1162] https://github.com/STEllAR-GROUP/hpx/issues/2324
[1163] https://github.com/STEllAR-GROUP/hpx/issues/2266
[1164] https://github.com/STEllAR-GROUP/hpx/pull/2195
[1165] https://github.com/STEllAR-GROUP/hpx/issues/2137
[1166] https://github.com/STEllAR-GROUP/hpx/issues/2092
[1167] https://github.com/STEllAR-GROUP/hpx/issues/2026
[1168] https://github.com/STEllAR-GROUP/hpx/issues/1932
[1169] https://github.com/STEllAR-GROUP/hpx/pull/1914
[1170] https://github.com/STEllAR-GROUP/hpx/issues/1598
[1171] https://github.com/STEllAR-GROUP/hpx/issues/1404

- Issue #1400[1172] - hpx::util::unwrapped doesn't work with non-future types
- Issue #1205[1173] - TSS is broken
- Issue #1126[1174] - vector<future<T> > does not work gracefully with dataflow, when_all and unwrapped
- Issue #1056[1175] - Thread manager cleanup
- Issue #863[1176] - Futures should not require a default constructor
- Issue #856[1177] - Allow runtimemode_connect to be used with security enabled
- Issue #726[1178] - Valgrind
- Issue #701[1179] - Add RCR performance counter component
- Issue #528[1180] - Add support for known failures and warning count/comparisons to hpx_run_tests.py

2.10.5 *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 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 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[1181] for more details).
- Added a new facility `hpx::lcos::barrier`, replacing the equally named older one. The new facility has a slightly changed API and is much more efficient. Most notable, the new facility exposes a (global) function `hpx::lcos::barrier::synchronize()` which represents a global barrier across all localities.
- We have started to add support for vectorization to our parallel algorithm implementations. This support depends on using an external library, currently either Vc Library or Boost.SIMD[1182]. Please see Issue #2333[1183] 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[1184] 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.

[1172] https://github.com/STEllAR-GROUP/hpx/issues/1400
[1173] https://github.com/STEllAR-GROUP/hpx/issues/1205
[1174] https://github.com/STEllAR-GROUP/hpx/issues/1126
[1175] https://github.com/STEllAR-GROUP/hpx/issues/1056
[1176] https://github.com/STEllAR-GROUP/hpx/issues/863
[1177] https://github.com/STEllAR-GROUP/hpx/issues/856
[1178] https://github.com/STEllAR-GROUP/hpx/issues/726
[1179] https://github.com/STEllAR-GROUP/hpx/issues/701
[1180] https://github.com/STEllAR-GROUP/hpx/issues/528
[1181] https://github.com/STEllAR-GROUP/hpx/pull/2289
[1182] https://github.com/NumScale/boost.simd
[1183] https://github.com/STEllAR-GROUP/hpx/issues/2333
[1184] http://stellar-group.org/2016/09/vectorized-cpp-parallel-algorithms-with-hpx/

- 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 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.
- 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[1185] - Adding apex data
- PR #2595[1186] - Remove obsolete file

[1185] https://github.com/STEllAR-GROUP/hpx/pull/2596
[1186] https://github.com/STEllAR-GROUP/hpx/pull/2595

- Issue #2594[1187] - FindOpenCL.cmake mismatch with the official cmake module
- PR #2592[1188] - First attempt to introduce spmd_block in hpx
- Issue #2591[1189] - Feature request: continuation (then) which does not require the callable object to take a future<R> as parameter
- PR #2588[1190] - Daint fixes
- PR #2587[1191] - Fixing transfer_(continuation)_action::schedule
- PR #2585[1192] - Work around MSVC having an ICE when compiling with -Ob2
- PR #2583[1193] - chaning 7zip command to 7za in roll_release.sh
- PR #2582[1194] - First attempt to introduce spmd_block in hpx
- PR #2581[1195] - Enable annotated function for parallel algorithms
- PR #2580[1196] - First attempt to introduce spmd_block in hpx
- PR #2579[1197] - Make thread NICE level setting an option
- PR #2578[1198] - Implementing enqueue instead of busy wait when no sender is available
- PR #2577[1199] - Retrieve -std=c++11 consistent nvcc flag
- PR #2576[1200] - Add missing dependencies of cuda based tests
- PR #2575[1201] - Remove warnings due to some captured variables
- PR #2573[1202] - Attempt to resolve resolve_locality
- PR #2572[1203] - Adding APEX hooks to background thread
- PR #2571[1204] - Pick up hpx.ignore_batch_env from config map
- PR #2570[1205] - Add commandline options –hpx:print-counters-locally
- PR #2569[1206] - Fix computeapi unit tests
- PR #2567[1207] - This adds another barrier::synchronize before registering performance counters
- PR #2564[1208] - Cray static toolchain support
- PR #2563[1209] - Fixed unhandled exception during startup

[1187] https://github.com/STEllAR-GROUP/hpx/issues/2594
[1188] https://github.com/STEllAR-GROUP/hpx/pull/2592
[1189] https://github.com/STEllAR-GROUP/hpx/issues/2591
[1190] https://github.com/STEllAR-GROUP/hpx/pull/2588
[1191] https://github.com/STEllAR-GROUP/hpx/pull/2587
[1192] https://github.com/STEllAR-GROUP/hpx/pull/2585
[1193] https://github.com/STEllAR-GROUP/hpx/pull/2583
[1194] https://github.com/STEllAR-GROUP/hpx/pull/2582
[1195] https://github.com/STEllAR-GROUP/hpx/pull/2581
[1196] https://github.com/STEllAR-GROUP/hpx/pull/2580
[1197] https://github.com/STEllAR-GROUP/hpx/pull/2579
[1198] https://github.com/STEllAR-GROUP/hpx/pull/2578
[1199] https://github.com/STEllAR-GROUP/hpx/pull/2577
[1200] https://github.com/STEllAR-GROUP/hpx/pull/2576
[1201] https://github.com/STEllAR-GROUP/hpx/pull/2575
[1202] https://github.com/STEllAR-GROUP/hpx/pull/2573
[1203] https://github.com/STEllAR-GROUP/hpx/pull/2572
[1204] https://github.com/STEllAR-GROUP/hpx/pull/2571
[1205] https://github.com/STEllAR-GROUP/hpx/pull/2570
[1206] https://github.com/STEllAR-GROUP/hpx/pull/2569
[1207] https://github.com/STEllAR-GROUP/hpx/pull/2567
[1208] https://github.com/STEllAR-GROUP/hpx/pull/2564
[1209] https://github.com/STEllAR-GROUP/hpx/pull/2563

- PR #2562[1210] - Remove partitioned_vector.cu from build tree when nvcc is used
- Issue #2561[1211] - octo-tiger crash with commit 6e921495ff6c26f125d62629cbaad0525f14f7ab
- PR #2560[1212] - Prevent -Wundef warnings on Vc version checks
- PR #2559[1213] - Allowing CUDA callback to set the future directly from an OS thread
- PR #2558[1214] - Remove warnings due to float precisions
- PR #2557[1215] - Removing bogus handling of compile flags for CUDA
- PR #2556[1216] - Fixing scan partitioner
- PR #2554[1217] - Add more diagnostics to error thrown from find_appropriate_destination
- Issue #2555[1218] - No valid parcelport configured
- PR #2553[1219] - Add cmake cuda_arch option
- PR #2552[1220] - Remove incomplete datapar bindings to libflatarray
- PR #2551[1221] - Rename hwloc_topology to hwloc_topology_info
- PR #2550[1222] - Apex api updates
- PR #2549[1223] - Pre-include defines.hpp to get the macro HPX_HAVE_CUDA value
- PR #2548[1224] - Fixing issue with disconnect
- PR #2546[1225] - Some fixes around cuda clang partitioned_vector example
- PR #2545[1226] - Fix uses of the Vc2 datapar flags; the value, not the type, should be passed to functions
- PR #2542[1227] - Make HPX_WITH_MALLOC easier to use
- PR #2541[1228] - avoid recompiles when enabling/disabling examples
- PR #2540[1229] - Fixing usage of target_link_libraries()
- PR #2539[1230] - fix RPATH behaviour
- Issue #2538[1231] - HPX_WITH_CUDA corrupts compilation flags
- PR #2537[1232] - Add output of a Bazel Skylark extension for paths and compile options

[1210] https://github.com/STEllAR-GROUP/hpx/pull/2562
[1211] https://github.com/STEllAR-GROUP/hpx/issues/2561
[1212] https://github.com/STEllAR-GROUP/hpx/pull/2560
[1213] https://github.com/STEllAR-GROUP/hpx/pull/2559
[1214] https://github.com/STEllAR-GROUP/hpx/pull/2558
[1215] https://github.com/STEllAR-GROUP/hpx/pull/2557
[1216] https://github.com/STEllAR-GROUP/hpx/pull/2556
[1217] https://github.com/STEllAR-GROUP/hpx/pull/2554
[1218] https://github.com/STEllAR-GROUP/hpx/issues/2555
[1219] https://github.com/STEllAR-GROUP/hpx/pull/2553
[1220] https://github.com/STEllAR-GROUP/hpx/pull/2552
[1221] https://github.com/STEllAR-GROUP/hpx/pull/2551
[1222] https://github.com/STEllAR-GROUP/hpx/pull/2550
[1223] https://github.com/STEllAR-GROUP/hpx/pull/2549
[1224] https://github.com/STEllAR-GROUP/hpx/pull/2548
[1225] https://github.com/STEllAR-GROUP/hpx/pull/2546
[1226] https://github.com/STEllAR-GROUP/hpx/pull/2545
[1227] https://github.com/STEllAR-GROUP/hpx/pull/2542
[1228] https://github.com/STEllAR-GROUP/hpx/pull/2541
[1229] https://github.com/STEllAR-GROUP/hpx/pull/2540
[1230] https://github.com/STEllAR-GROUP/hpx/pull/2539
[1231] https://github.com/STEllAR-GROUP/hpx/issues/2538
[1232] https://github.com/STEllAR-GROUP/hpx/pull/2537

- PR #2536[1233] - Add counter exposing total available memory to Windows as well
- PR #2535[1234] - Remove obsolete support for security
- Issue #2534[1235] - Remove command line option `--hpx:run-agas-server`
- PR #2533[1236] - Pre-cache locality endpoints during bootstrap
- PR #2532[1237] - Fixing handling of GIDs during serialization preprocessing
- PR #2531[1238] - Amend uses of the term "functor"
- PR #2529[1239] - added counter for reading available memory
- PR #2527[1240] - Facilities to create actions from lambdas
- PR #2526[1241] - Updated docs: HPX_WITH_EXAMPLES
- PR #2525[1242] - Remove warnings related to unused captured variables
- Issue #2524[1243] - CMAKE failed because it is missing: TCMALLOC_LIBRARY TCMALLOC_INCLUDE_DIR
- PR #2523[1244] - Fixing compose_cb stack overflow
- PR #2522[1245] - Instead of unlocking, ignore the lock while creating the message handler
- PR #2521[1246] - Create `LPROGRESS_` logging macro to simplify progress tracking and timings
- PR #2520[1247] - Intel 17 support
- PR #2519[1248] - Fix components example
- PR #2518[1249] - Fixing parcel scheduling
- Issue #2517[1250] - Race condition during Parcel Coalescing Handler creation
- Issue #2516[1251] - HPX locks up when using at least 256 localities
- Issue #2515[1252] - error: Install cannot find "/lib/hpx/libparcel_coalescing.so.0.9.99" but I can see that file
- PR #2514[1253] - Making sure that all continuations of a shared_future are invoked in order
- PR #2513[1254] - Fixing locks held during suspension
- PR #2512[1255] - MPI Parcelport improvements and fixes related to the background work changes

1233 https://github.com/STEllAR-GROUP/hpx/pull/2536
1234 https://github.com/STEllAR-GROUP/hpx/pull/2535
1235 https://github.com/STEllAR-GROUP/hpx/issues/2534
1236 https://github.com/STEllAR-GROUP/hpx/pull/2533
1237 https://github.com/STEllAR-GROUP/hpx/pull/2532
1238 https://github.com/STEllAR-GROUP/hpx/pull/2531
1239 https://github.com/STEllAR-GROUP/hpx/pull/2529
1240 https://github.com/STEllAR-GROUP/hpx/pull/2527
1241 https://github.com/STEllAR-GROUP/hpx/pull/2526
1242 https://github.com/STEllAR-GROUP/hpx/pull/2525
1243 https://github.com/STEllAR-GROUP/hpx/issues/2524
1244 https://github.com/STEllAR-GROUP/hpx/pull/2523
1245 https://github.com/STEllAR-GROUP/hpx/pull/2522
1246 https://github.com/STEllAR-GROUP/hpx/pull/2521
1247 https://github.com/STEllAR-GROUP/hpx/pull/2520
1248 https://github.com/STEllAR-GROUP/hpx/pull/2519
1249 https://github.com/STEllAR-GROUP/hpx/pull/2518
1250 https://github.com/STEllAR-GROUP/hpx/issues/2517
1251 https://github.com/STEllAR-GROUP/hpx/issues/2516
1252 https://github.com/STEllAR-GROUP/hpx/issues/2515
1253 https://github.com/STEllAR-GROUP/hpx/pull/2514
1254 https://github.com/STEllAR-GROUP/hpx/pull/2513
1255 https://github.com/STEllAR-GROUP/hpx/pull/2512

- PR #2511[1256] - Fixing bit-wise (zero-copy) serialization
- Issue #2509[1257] - Linking errors in hwloc_topology
- PR #2508[1258] - Added documentation for debugging with core files
- PR #2506[1259] - Fixing background work invocations
- PR #2505[1260] - Fix tuple serialization
- Issue #2504[1261] - Ensure continuations are called in the order they have been attached
- PR #2503[1262] - Adding serialization support for Vc v2 (datapar)
- PR #2502[1263] - Resolve various, minor compiler warnings
- PR #2501[1264] - Some other fixes around cuda examples
- Issue #2500[1265] - nvcc / cuda clang issue due to a missing -DHPX_WITH_CUDA flag
- PR #2499[1266] - Adding support for std::array to wait_all and friends
- PR #2498[1267] - Execute background work as HPX thread
- PR #2497[1268] - Fixing configuration options for spinlock-deadlock detection
- PR #2496[1269] - Accounting for different compilers in CrayKNL toolchain file
- PR #2494[1270] - Adding component base class which ties a component instance to a given executor
- PR #2493[1271] - Enable controlling amount of pending threads which must be available to allow thread stealing
- PR #2492[1272] - Adding new command line option –hpx:print-counter-reset
- PR #2491[1273] - Resolve ambiguities when compiling with APEX
- PR #2490[1274] - Resuming threads waiting on future with higher priority
- Issue #2489[1275] - nvcc issue because -std=c++11 appears twice
- PR #2488[1276] - Adding performance counters exposing the internal idle and busy-loop counters
- PR #2487[1277] - Allowing for plain suspend to reschedule thread right away
- PR #2486[1278] - Only flag HPX code for CUDA if HPX_WITH_CUDA is set

[1256] https://github.com/STEllAR-GROUP/hpx/pull/2511
[1257] https://github.com/STEllAR-GROUP/hpx/issues/2509
[1258] https://github.com/STEllAR-GROUP/hpx/pull/2508
[1259] https://github.com/STEllAR-GROUP/hpx/pull/2506
[1260] https://github.com/STEllAR-GROUP/hpx/pull/2505
[1261] https://github.com/STEllAR-GROUP/hpx/issues/2504
[1262] https://github.com/STEllAR-GROUP/hpx/pull/2503
[1263] https://github.com/STEllAR-GROUP/hpx/pull/2502
[1264] https://github.com/STEllAR-GROUP/hpx/pull/2501
[1265] https://github.com/STEllAR-GROUP/hpx/issues/2500
[1266] https://github.com/STEllAR-GROUP/hpx/pull/2499
[1267] https://github.com/STEllAR-GROUP/hpx/pull/2498
[1268] https://github.com/STEllAR-GROUP/hpx/pull/2497
[1269] https://github.com/STEllAR-GROUP/hpx/pull/2496
[1270] https://github.com/STEllAR-GROUP/hpx/pull/2494
[1271] https://github.com/STEllAR-GROUP/hpx/pull/2493
[1272] https://github.com/STEllAR-GROUP/hpx/pull/2492
[1273] https://github.com/STEllAR-GROUP/hpx/pull/2491
[1274] https://github.com/STEllAR-GROUP/hpx/pull/2490
[1275] https://github.com/STEllAR-GROUP/hpx/issues/2489
[1276] https://github.com/STEllAR-GROUP/hpx/pull/2488
[1277] https://github.com/STEllAR-GROUP/hpx/pull/2487
[1278] https://github.com/STEllAR-GROUP/hpx/pull/2486

- PR #2485[1279] - Making thread-queue parameters runtime-configurable
- PR #2484[1280] - Added atomic counter for parcel-destinations
- PR #2483[1281] - Added priority-queue lifo scheduler
- PR #2482[1282] - Changing scheduler to steal only if more than a minimal number of tasks are available
- PR #2481[1283] - Extending command line option –hpx:print-counter-destination to support value ‘none’
- PR #2479[1284] - Added option to disable signal handler
- PR #2478[1285] - Making sure the sine performance counter module gets loaded only for the corresponding example
- Issue #2477[1286] - Breaking at a throw statement
- PR #2476[1287] - Annotated function
- PR #2475[1288] - Ensure that using %osthread% during logging will not throw for non-hpx threads
- PR #2474[1289] - Remove now superficial non_direct actions from base_lco and friends
- PR #2473[1290] - Refining support for ITTNotify
- PR #2472[1291] - Some fixes around hpx compute
- Issue #2470[1292] - redefinition of boost::detail::spinlock
- Issue #2469[1293] - Dataflow performance issue
- PR #2468[1294] - Perf docs update
- PR #2466[1295] - Guarantee to execute remote direct actions on HPX-thread
- PR #2465[1296] - Improve demo : Async copy and fixed device handling
- PR #2464[1297] - Adding performance counter exposing instantaneous scheduler utilization
- PR #2463[1298] - Downcast to future<void>
- PR #2462[1299] - Fixed usage of ITT-Notify API with Intel Amplifier
- PR #2461[1300] - Cublas demo
- PR #2460[1301] - Fixing thread bindings

1279 https://github.com/STEllAR-GROUP/hpx/pull/2485
1280 https://github.com/STEllAR-GROUP/hpx/pull/2484
1281 https://github.com/STEllAR-GROUP/hpx/pull/2483
1282 https://github.com/STEllAR-GROUP/hpx/pull/2482
1283 https://github.com/STEllAR-GROUP/hpx/pull/2481
1284 https://github.com/STEllAR-GROUP/hpx/pull/2479
1285 https://github.com/STEllAR-GROUP/hpx/pull/2478
1286 https://github.com/STEllAR-GROUP/hpx/issues/2477
1287 https://github.com/STEllAR-GROUP/hpx/pull/2476
1288 https://github.com/STEllAR-GROUP/hpx/pull/2475
1289 https://github.com/STEllAR-GROUP/hpx/pull/2474
1290 https://github.com/STEllAR-GROUP/hpx/pull/2473
1291 https://github.com/STEllAR-GROUP/hpx/pull/2472
1292 https://github.com/STEllAR-GROUP/hpx/issues/2470
1293 https://github.com/STEllAR-GROUP/hpx/issues/2469
1294 https://github.com/STEllAR-GROUP/hpx/pull/2468
1295 https://github.com/STEllAR-GROUP/hpx/pull/2466
1296 https://github.com/STEllAR-GROUP/hpx/pull/2465
1297 https://github.com/STEllAR-GROUP/hpx/pull/2464
1298 https://github.com/STEllAR-GROUP/hpx/pull/2463
1299 https://github.com/STEllAR-GROUP/hpx/pull/2462
1300 https://github.com/STEllAR-GROUP/hpx/pull/2461
1301 https://github.com/STEllAR-GROUP/hpx/pull/2460

- PR #2459[1302] - Make -std=c++11 nvcc flag consistent for in-build and installed versions
- Issue #2457[1303] - Segmentation fault when registering a partitioned vector
- PR #2452[1304] - Properly releasing global barrier for unhandled exceptions
- PR #2451[1305] - Fixing long shutdown times
- PR #2450[1306] - Attempting to fix initialization errors on newer platforms (Boost V1.63)
- PR #2449[1307] - Replace BOOST_COMPILER_FENCE with an HPX version
- PR #2448[1308] - This fixes a possible race in the migration code
- **PR #2445[1309] - Fixing dataflow et.al. for futures or future-ranges wrapped** into ref()
- PR #2444[1310] - Fix segfaults
- PR #2443[1311] - Issue 2442
- Issue #2442[1312] - Mismatch between #if/#endif and namespace scope brackets in this_thread_executers.hpp
- Issue #2441[1313] - undeclared identifier BOOST_COMPILER_FENCE
- PR #2440[1314] - Knl build
- PR #2438[1315] - Datapar backend
- PR #2437[1316] - Adapt algorithm parameter sequence changes from C++17
- PR #2436[1317] - Adapt execution policy name changes from C++17
- Issue #2435[1318] - Trunk broken, undefined reference to hpx::thread::interrupt(hpx::thread::id, bool)
- PR #2434[1319] - More fixes to resource manager
- PR #2433[1320] - Added versions of `hpx::get_ptr` taking client side representations
- PR #2432[1321] - Warning fixes
- PR #2431[1322] - Adding facility representing set of performance counters
- PR #2430[1323] - Fix parallel_executor thread spawning
- PR #2429[1324] - Fix attribute warning for gcc

[1302] https://github.com/STEllAR-GROUP/hpx/pull/2459
[1303] https://github.com/STEllAR-GROUP/hpx/issues/2457
[1304] https://github.com/STEllAR-GROUP/hpx/pull/2452
[1305] https://github.com/STEllAR-GROUP/hpx/pull/2451
[1306] https://github.com/STEllAR-GROUP/hpx/pull/2450
[1307] https://github.com/STEllAR-GROUP/hpx/pull/2449
[1308] https://github.com/STEllAR-GROUP/hpx/pull/2448
[1309] https://github.com/STEllAR-GROUP/hpx/pull/2445
[1310] https://github.com/STEllAR-GROUP/hpx/pull/2444
[1311] https://github.com/STEllAR-GROUP/hpx/pull/2443
[1312] https://github.com/STEllAR-GROUP/hpx/issues/2442
[1313] https://github.com/STEllAR-GROUP/hpx/issues/2441
[1314] https://github.com/STEllAR-GROUP/hpx/pull/2440
[1315] https://github.com/STEllAR-GROUP/hpx/pull/2438
[1316] https://github.com/STEllAR-GROUP/hpx/pull/2437
[1317] https://github.com/STEllAR-GROUP/hpx/pull/2436
[1318] https://github.com/STEllAR-GROUP/hpx/issues/2435
[1319] https://github.com/STEllAR-GROUP/hpx/pull/2434
[1320] https://github.com/STEllAR-GROUP/hpx/pull/2433
[1321] https://github.com/STEllAR-GROUP/hpx/pull/2432
[1322] https://github.com/STEllAR-GROUP/hpx/pull/2431
[1323] https://github.com/STEllAR-GROUP/hpx/pull/2430
[1324] https://github.com/STEllAR-GROUP/hpx/pull/2429

- Issue #2427[1325] - Seg fault running octo-tiger with latest HPX commit
- Issue #2426[1326] - Bug in 9592f5c0bc29806fce0dbe73f35b6ca7e027edcb causes immediate crash in Octo-tiger
- PR #2425[1327] - Fix nvcc errors due to constexpr specifier
- Issue #2424[1328] - Async action on component present on hpx::find_here is executing synchronously
- PR #2423[1329] - Fix nvcc errors due to constexpr specifier
- PR #2422[1330] - Implementing hpx::this_thread thread data functions
- PR #2421[1331] - Adding benchmark for wait_all
- Issue #2420[1332] - Returning object of a component client from another component action fails
- PR #2419[1333] - Infiniband parcelport
- Issue #2418[1334] - gcc + nvcc fails to compile code that uses partitioned_vector
- PR #2417[1335] - Fixing context switching
- PR #2416[1336] - Adding fixes and workarounds to allow compilation with nvcc/msvc (VS2015up3)
- PR #2415[1337] - Fix errors coming from hpx compute examples
- PR #2414[1338] - Fixing msvc12
- PR #2413[1339] - Enable cuda/nvcc or cuda/clang when using add_hpx_executable()
- PR #2412[1340] - Fix issue in HPX_SetupTarget.cmake when cuda is used
- PR #2411[1341] - This fixes the core compilation issues with MSVC12
- Issue #2410[1342] - `undefined reference to opal_hwloc191_hwloc_.....`
- PR #2409[1343] - Fixing locking for channel and receive_buffer
- PR #2407[1344] - Solving #2402 and #2403
- PR #2406[1345] - Improve guards
- PR #2405[1346] - Enable parallel::for_each for iterators returning proxy types
- PR #2404[1347] - Forward the explicitly given result_type in the hpx invoke

1325 https://github.com/STEllAR-GROUP/hpx/issues/2427
1326 https://github.com/STEllAR-GROUP/hpx/issues/2426
1327 https://github.com/STEllAR-GROUP/hpx/pull/2425
1328 https://github.com/STEllAR-GROUP/hpx/issues/2424
1329 https://github.com/STEllAR-GROUP/hpx/pull/2423
1330 https://github.com/STEllAR-GROUP/hpx/pull/2422
1331 https://github.com/STEllAR-GROUP/hpx/pull/2421
1332 https://github.com/STEllAR-GROUP/hpx/issues/2420
1333 https://github.com/STEllAR-GROUP/hpx/pull/2419
1334 https://github.com/STEllAR-GROUP/hpx/issues/2418
1335 https://github.com/STEllAR-GROUP/hpx/pull/2417
1336 https://github.com/STEllAR-GROUP/hpx/pull/2416
1337 https://github.com/STEllAR-GROUP/hpx/pull/2415
1338 https://github.com/STEllAR-GROUP/hpx/pull/2414
1339 https://github.com/STEllAR-GROUP/hpx/pull/2413
1340 https://github.com/STEllAR-GROUP/hpx/pull/2412
1341 https://github.com/STEllAR-GROUP/hpx/pull/2411
1342 https://github.com/STEllAR-GROUP/hpx/issues/2410
1343 https://github.com/STEllAR-GROUP/hpx/pull/2409
1344 https://github.com/STEllAR-GROUP/hpx/pull/2407
1345 https://github.com/STEllAR-GROUP/hpx/pull/2406
1346 https://github.com/STEllAR-GROUP/hpx/pull/2405
1347 https://github.com/STEllAR-GROUP/hpx/pull/2404

- Issue #2403[1348] - datapar_execution + zip iterator: lambda arguments aren't references
- Issue #2402[1349] - datapar algorithm instantiated with wrong type #2402
- PR #2401[1350] - Added support for imported libraries to HPX_Libraries.cmake
- PR #2400[1351] - Use CMake policy CMP0060
- Issue #2399[1352] - Error trying to push back vector of futures to vector
- PR #2398[1353] - Allow config #defines to be written out to custom config/defines.hpp
- Issue #2397[1354] - CMake generated config defines can cause tedious rebuilds category
- Issue #2396[1355] - BOOST_ROOT paths are not used at link time
- PR #2395[1356] - Fix target_link_libraries() issue when HPX Cuda is enabled
- Issue #2394[1357] - Template compilation error using HPX_WITH_DATAPAR_LIBFLATARRAY
- PR #2393[1358] - Fixing lock registration for recursive mutex
- PR #2392[1359] - Add keywords in target_link_libraries in hpx_setup_target
- PR #2391[1360] - Clang goroutines
- Issue #2390[1361] - Adapt execution policy name changes from C++17
- PR #2389[1362] - Chunk allocator and pool are not used and are obsolete
- PR #2388[1363] - Adding functionalities to datapar needed by octotiger
- PR #2387[1364] - Fixing race condition for early parcels
- Issue #2386[1365] - Lock registration broken for recursive_mutex
- PR #2385[1366] - Datapar zip iterator
- PR #2384[1367] - Fixing race condition in for_loop_reduction
- PR #2383[1368] - Continuations
- PR #2382[1369] - add LibFlatArray-based backend for datapar
- PR #2381[1370] - remove unused typedef to get rid of compiler warnings

[1348] https://github.com/STEllAR-GROUP/hpx/issues/2403
[1349] https://github.com/STEllAR-GROUP/hpx/issues/2402
[1350] https://github.com/STEllAR-GROUP/hpx/pull/2401
[1351] https://github.com/STEllAR-GROUP/hpx/pull/2400
[1352] https://github.com/STEllAR-GROUP/hpx/issues/2399
[1353] https://github.com/STEllAR-GROUP/hpx/pull/2398
[1354] https://github.com/STEllAR-GROUP/hpx/issues/2397
[1355] https://github.com/STEllAR-GROUP/hpx/issues/2396
[1356] https://github.com/STEllAR-GROUP/hpx/pull/2395
[1357] https://github.com/STEllAR-GROUP/hpx/issues/2394
[1358] https://github.com/STEllAR-GROUP/hpx/pull/2393
[1359] https://github.com/STEllAR-GROUP/hpx/pull/2392
[1360] https://github.com/STEllAR-GROUP/hpx/pull/2391
[1361] https://github.com/STEllAR-GROUP/hpx/issues/2390
[1362] https://github.com/STEllAR-GROUP/hpx/pull/2389
[1363] https://github.com/STEllAR-GROUP/hpx/pull/2388
[1364] https://github.com/STEllAR-GROUP/hpx/pull/2387
[1365] https://github.com/STEllAR-GROUP/hpx/issues/2386
[1366] https://github.com/STEllAR-GROUP/hpx/pull/2385
[1367] https://github.com/STEllAR-GROUP/hpx/pull/2384
[1368] https://github.com/STEllAR-GROUP/hpx/pull/2383
[1369] https://github.com/STEllAR-GROUP/hpx/pull/2382
[1370] https://github.com/STEllAR-GROUP/hpx/pull/2381

- PR #2380[1371] - Tau cleanup
- PR #2379[1372] - Can send immediate
- PR #2378[1373] - Renaming copy_helper/copy_n_helper/move_helper/move_n_helper
- Issue #2376[1374] - Boost trunk's spinlock initializer fails to compile
- PR #2375[1375] - Add support for minimal thread local data
- PR #2374[1376] - Adding API functions set_config_entry_callback
- PR #2373[1377] - Add a simple utility for debugging that gives supended task backtraces
- PR #2372[1378] - Barrier Fixes
- Issue #2370[1379] - Can't wait on a wrapped future
- PR #2369[1380] - Fixing stable_partition
- PR #2367[1381] - Fixing find_prefixes for Windows platforms
- PR #2366[1382] - Testing for experimental/optional only in C++14 mode
- PR #2364[1383] - Adding set_config_entry
- PR #2363[1384] - Fix papi
- PR #2362[1385] - Adding missing macros for new non-direct actions
- PR #2361[1386] - Improve cmake output to help debug compiler incompatibility check
- PR #2360[1387] - Fixing race condition in condition_variable
- PR #2359[1388] - Fixing shutdown when parcels are still in flight
- Issue #2357[1389] - failed to insert console_print_action into typename_to_id_t registry
- PR #2356[1390] - Fixing return type of get_iterator_tuple
- PR #2355[1391] - Fixing compilation against Boost 1 62
- PR #2354[1392] - Adding serialization for mask_type if CPU_COUNT > 64
- PR #2353[1393] - Adding hooks to tie in APEX into the parcel layer

1371 https://github.com/STEllAR-GROUP/hpx/pull/2380
1372 https://github.com/STEllAR-GROUP/hpx/pull/2379
1373 https://github.com/STEllAR-GROUP/hpx/pull/2378
1374 https://github.com/STEllAR-GROUP/hpx/issues/2376
1375 https://github.com/STEllAR-GROUP/hpx/pull/2375
1376 https://github.com/STEllAR-GROUP/hpx/pull/2374
1377 https://github.com/STEllAR-GROUP/hpx/pull/2373
1378 https://github.com/STEllAR-GROUP/hpx/pull/2372
1379 https://github.com/STEllAR-GROUP/hpx/issues/2370
1380 https://github.com/STEllAR-GROUP/hpx/pull/2369
1381 https://github.com/STEllAR-GROUP/hpx/pull/2367
1382 https://github.com/STEllAR-GROUP/hpx/pull/2366
1383 https://github.com/STEllAR-GROUP/hpx/pull/2364
1384 https://github.com/STEllAR-GROUP/hpx/pull/2363
1385 https://github.com/STEllAR-GROUP/hpx/pull/2362
1386 https://github.com/STEllAR-GROUP/hpx/pull/2361
1387 https://github.com/STEllAR-GROUP/hpx/pull/2360
1388 https://github.com/STEllAR-GROUP/hpx/pull/2359
1389 https://github.com/STEllAR-GROUP/hpx/issues/2357
1390 https://github.com/STEllAR-GROUP/hpx/pull/2356
1391 https://github.com/STEllAR-GROUP/hpx/pull/2355
1392 https://github.com/STEllAR-GROUP/hpx/pull/2354
1393 https://github.com/STEllAR-GROUP/hpx/pull/2353

- Issue #2352[1394] - Compile errors when using intel 17 beta (for KNL) on edison
- PR #2351[1395] - Fix function vtable get_function_address implementation
- Issue #2350[1396] - Build failure - master branch (4de09f5) with Intel Compiler v17
- PR #2349[1397] - Enabling zero-copy serialization support for std::vector<>
- PR #2348[1398] - Adding test to verify #2334 is fixed
- PR #2347[1399] - Bug fixes for hpx.compute and hpx::lcos::channel
- PR #2346[1400] - Removing cmake "find" files that are in the APEX cmake Modules
- PR #2345[1401] - Implemented parallel::stable_partition
- PR #2344[1402] - Making hpx::lcos::channel usable with basename registration
- PR #2343[1403] - Fix a couple of examples that failed to compile after recent api changes
- Issue #2342[1404] - Enabling APEX causes link errors
- PR #2341[1405] - Removing cmake "find" files that are in the APEX cmake Modules
- PR #2340[1406] - Implemented all existing datapar algorithms using Boost.SIMD
- PR #2339[1407] - Fixing 2338
- PR #2338[1408] - Possible race in sliding semaphore
- PR #2337[1409] - Adjust osu_latency test to measure window_size parcels in flight at once
- PR #2336[1410] - Allowing remote direct actions to be executed without spawning a task
- PR #2335[1411] - Making sure multiple components are properly initialized from arguments
- Issue #2334[1412] - Cannot construct component with large vector on a remote locality
- PR #2332[1413] - Fixing hpx::lcos::local::barrier
- PR #2331[1414] - Updating APEX support to include OTF2
- PR #2330[1415] - Support for data-parallelism for parallel algorithms
- Issue #2329[1416] - Coordinate settings in cmake

[1394] https://github.com/STEllAR-GROUP/hpx/issues/2352
[1395] https://github.com/STEllAR-GROUP/hpx/pull/2351
[1396] https://github.com/STEllAR-GROUP/hpx/issues/2350
[1397] https://github.com/STEllAR-GROUP/hpx/pull/2349
[1398] https://github.com/STEllAR-GROUP/hpx/pull/2348
[1399] https://github.com/STEllAR-GROUP/hpx/pull/2347
[1400] https://github.com/STEllAR-GROUP/hpx/pull/2346
[1401] https://github.com/STEllAR-GROUP/hpx/pull/2345
[1402] https://github.com/STEllAR-GROUP/hpx/pull/2344
[1403] https://github.com/STEllAR-GROUP/hpx/pull/2343
[1404] https://github.com/STEllAR-GROUP/hpx/issues/2342
[1405] https://github.com/STEllAR-GROUP/hpx/pull/2341
[1406] https://github.com/STEllAR-GROUP/hpx/pull/2340
[1407] https://github.com/STEllAR-GROUP/hpx/pull/2339
[1408] https://github.com/STEllAR-GROUP/hpx/pull/2338
[1409] https://github.com/STEllAR-GROUP/hpx/pull/2337
[1410] https://github.com/STEllAR-GROUP/hpx/pull/2336
[1411] https://github.com/STEllAR-GROUP/hpx/pull/2335
[1412] https://github.com/STEllAR-GROUP/hpx/issues/2334
[1413] https://github.com/STEllAR-GROUP/hpx/pull/2332
[1414] https://github.com/STEllAR-GROUP/hpx/pull/2331
[1415] https://github.com/STEllAR-GROUP/hpx/pull/2330
[1416] https://github.com/STEllAR-GROUP/hpx/issues/2329

- PR #2328[1417] - fix LibGeoDecomp builds with HPX + GCC 5.3.0 + CUDA 8RC
- PR #2326[1418] - Making scan_partitioner work (for now)
- Issue #2323[1419] - Constructing a vector of components only correctly initializes the first component
- PR #2322[1420] - Fix problems that bubbled up after merging #2278
- PR #2321[1421] - Scalable barrier
- PR #2320[1422] - Std flag fixes
- Issue #2319[1423] - -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[1424] - Improve handling of –hpx:bind=<bind-spec>
- PR #2317[1425] - Making sure command line warnings are printed once only
- PR #2316[1426] - Fixing command line handling for default bind mode
- PR #2315[1427] - Set id_retrieved if set_id is present
- Issue #2314[1428] - Warning for requested/allocated thread discrepancy is printed twice
- Issue #2313[1429] - –hpx:print-bind doesn't work with –hpx:pu-step
- Issue #2312[1430] - –hpx:bind range specifier restrictions are overly restrictive
- Issue #2311[1431] - hpx_0.9.99 out of project build fails
- PR #2310[1432] - Simplify function registration
- PR #2309[1433] - Spelling and grammar revisions in documentation (and some code)
- PR #2306[1434] - Correct minor typo in the documentation
- PR #2305[1435] - Cleaning up and fixing parcel coalescing
- PR #2304[1436] - Inspect checks for stream related includes
- PR #2303[1437] - Add functionality allowing to enumerate threads of given state
- PR #2301[1438] - Algorithm overloads fix for VS2013
- PR #2300[1439] - Use <cstdint>, add inspect checks

1417 https://github.com/STEllAR-GROUP/hpx/pull/2328
1418 https://github.com/STEllAR-GROUP/hpx/pull/2326
1419 https://github.com/STEllAR-GROUP/hpx/issues/2323
1420 https://github.com/STEllAR-GROUP/hpx/pull/2322
1421 https://github.com/STEllAR-GROUP/hpx/pull/2321
1422 https://github.com/STEllAR-GROUP/hpx/pull/2320
1423 https://github.com/STEllAR-GROUP/hpx/issues/2319
1424 https://github.com/STEllAR-GROUP/hpx/pull/2318
1425 https://github.com/STEllAR-GROUP/hpx/pull/2317
1426 https://github.com/STEllAR-GROUP/hpx/pull/2316
1427 https://github.com/STEllAR-GROUP/hpx/pull/2315
1428 https://github.com/STEllAR-GROUP/hpx/issues/2314
1429 https://github.com/STEllAR-GROUP/hpx/issues/2313
1430 https://github.com/STEllAR-GROUP/hpx/issues/2312
1431 https://github.com/STEllAR-GROUP/hpx/issues/2311
1432 https://github.com/STEllAR-GROUP/hpx/pull/2310
1433 https://github.com/STEllAR-GROUP/hpx/pull/2309
1434 https://github.com/STEllAR-GROUP/hpx/pull/2306
1435 https://github.com/STEllAR-GROUP/hpx/pull/2305
1436 https://github.com/STEllAR-GROUP/hpx/pull/2304
1437 https://github.com/STEllAR-GROUP/hpx/pull/2303
1438 https://github.com/STEllAR-GROUP/hpx/pull/2301
1439 https://github.com/STEllAR-GROUP/hpx/pull/2300

- PR #2299[1440] - Replace boost::[c]ref with std::[c]ref, add inspect checks
- PR #2297[1441] - Fixing compilation with no hw_loc
- PR #2296[1442] - Hpx compute
- PR #2295[1443] - Making sure for_loop(execution::par, 0, N, …) is actually executed in parallel
- PR #2294[1444] - Throwing exceptions if the runtime is not up and running
- PR #2293[1445] - Removing unused parcel port code
- PR #2292[1446] - Refactor function vtables
- PR #2291[1447] - Fixing 2286
- PR #2290[1448] - Simplify algorithm overloads
- PR #2289[1449] - Adding performance counters reporting parcel related data on a per-action basis
- Issue #2288[1450] - Remove dormant parcelports
- Issue #2286[1451] - adjustments to parcel handling to support parcelports that do not need a connection cache
- PR #2285[1452] - add CMake option to disable package export
- PR #2283[1453] - Add more inspect checks for use of deprecated components
- Issue #2282[1454] - Arithmetic exception in executor static chunker
- Issue #2281[1455] - For loop doesn't parallelize
- PR #2280[1456] - Fixing 2277: build failure with PAPI
- PR #2279[1457] - Child vs parent stealing
- Issue #2277[1458] - master branch build failure (53c5b4f) with papi
- PR #2276[1459] - Compile time launch policies
- PR #2275[1460] - Replace boost::chrono with std::chrono in interfaces
- PR #2274[1461] - Replace most uses of Boost.Assign with initializer list
- PR #2273[1462] - Fixed typos

1440 https://github.com/STEllAR-GROUP/hpx/pull/2299
1441 https://github.com/STEllAR-GROUP/hpx/pull/2297
1442 https://github.com/STEllAR-GROUP/hpx/pull/2296
1443 https://github.com/STEllAR-GROUP/hpx/pull/2295
1444 https://github.com/STEllAR-GROUP/hpx/pull/2294
1445 https://github.com/STEllAR-GROUP/hpx/pull/2293
1446 https://github.com/STEllAR-GROUP/hpx/pull/2292
1447 https://github.com/STEllAR-GROUP/hpx/pull/2291
1448 https://github.com/STEllAR-GROUP/hpx/pull/2290
1449 https://github.com/STEllAR-GROUP/hpx/pull/2289
1450 https://github.com/STEllAR-GROUP/hpx/issues/2288
1451 https://github.com/STEllAR-GROUP/hpx/issues/2286
1452 https://github.com/STEllAR-GROUP/hpx/pull/2285
1453 https://github.com/STEllAR-GROUP/hpx/pull/2283
1454 https://github.com/STEllAR-GROUP/hpx/issues/2282
1455 https://github.com/STEllAR-GROUP/hpx/issues/2281
1456 https://github.com/STEllAR-GROUP/hpx/pull/2280
1457 https://github.com/STEllAR-GROUP/hpx/pull/2279
1458 https://github.com/STEllAR-GROUP/hpx/issues/2277
1459 https://github.com/STEllAR-GROUP/hpx/pull/2276
1460 https://github.com/STEllAR-GROUP/hpx/pull/2275
1461 https://github.com/STEllAR-GROUP/hpx/pull/2274
1462 https://github.com/STEllAR-GROUP/hpx/pull/2273

- PR #2272[1463] - Inspect checks
- PR #2270[1464] - Adding test verifying -Ihpx.os_threads=all
- PR #2269[1465] - Added inspect check for now obsolete boost type traits
- PR #2268[1466] - Moving more code into source files
- Issue #2267[1467] - Add inspect support to deprecate Boost.TypeTraits
- PR #2265[1468] - Adding channel LCO
- PR #2264[1469] - Make support for std::ref mandatory
- PR #2263[1470] - Constrain tuple_member forwarding constructor
- Issue #2262[1471] - Test hpx.os_threads=all
- Issue #2261[1472] - OS X: Error: no matching constructor for initialization of 'hpx::lcos::local::condition_variable_any'
- Issue #2260[1473] - Make support for std::ref mandatory
- PR #2259[1474] - Remove most of Boost.MPL, Boost.EnableIf and Boost.TypeTraits
- PR #2258[1475] - Fixing #2256
- PR #2257[1476] - Fixing launch process
- Issue #2256[1477] - Actions are not registered if not invoked
- PR #2255[1478] - Coalescing histogram
- PR #2254[1479] - Silence explicit initialization in copy-constructor warnings
- PR #2253[1480] - Drop support for GCC 4.6 and 4.7
- PR #2252[1481] - Prepare V1.0
- PR #2251[1482] - Convert to 0.9.99
- PR #2249[1483] - Adding iterator_facade and iterator_adaptor
- Issue #2248[1484] - Need a feature to yield to a new task immediately
- PR #2246[1485] - Adding split_future

1463 https://github.com/STEllAR-GROUP/hpx/pull/2272
1464 https://github.com/STEllAR-GROUP/hpx/pull/2270
1465 https://github.com/STEllAR-GROUP/hpx/pull/2269
1466 https://github.com/STEllAR-GROUP/hpx/pull/2268
1467 https://github.com/STEllAR-GROUP/hpx/issues/2267
1468 https://github.com/STEllAR-GROUP/hpx/pull/2265
1469 https://github.com/STEllAR-GROUP/hpx/pull/2264
1470 https://github.com/STEllAR-GROUP/hpx/pull/2263
1471 https://github.com/STEllAR-GROUP/hpx/issues/2262
1472 https://github.com/STEllAR-GROUP/hpx/issues/2261
1473 https://github.com/STEllAR-GROUP/hpx/issues/2260
1474 https://github.com/STEllAR-GROUP/hpx/pull/2259
1475 https://github.com/STEllAR-GROUP/hpx/pull/2258
1476 https://github.com/STEllAR-GROUP/hpx/pull/2257
1477 https://github.com/STEllAR-GROUP/hpx/issues/2256
1478 https://github.com/STEllAR-GROUP/hpx/pull/2255
1479 https://github.com/STEllAR-GROUP/hpx/pull/2254
1480 https://github.com/STEllAR-GROUP/hpx/pull/2253
1481 https://github.com/STEllAR-GROUP/hpx/pull/2252
1482 https://github.com/STEllAR-GROUP/hpx/pull/2251
1483 https://github.com/STEllAR-GROUP/hpx/pull/2249
1484 https://github.com/STEllAR-GROUP/hpx/issues/2248
1485 https://github.com/STEllAR-GROUP/hpx/pull/2246

- PR #2245[1486] - Add an example for handing over a component instance to a dynamically launched locality
- Issue #2243[1487] - Add example demonstrating AGAS symbolic name registration
- Issue #2242[1488] - pkgconfig test broken on CentOS 7 / Boost 1.61
- Issue #2241[1489] - Compilation error for partitioned vector in hpx_compute branch
- PR #2240[1490] - Fixing termination detection on one locality
- Issue #2239[1491] - Create a new facility lcos::split_all
- Issue #2236[1492] - hpx::cout vs. std::cout
- PR #2232[1493] - Implement local-only primary namespace service
- Issue #2147[1494] - would like to know how much data is being routed by particular actions
- Issue #2109[1495] - Warning while compiling hpx
- Issue #1973[1496] - Setting INTERFACE_COMPILE_OPTIONS for hpx_init in CMake taints Fortran_FLAGS
- Issue #1864[1497] - run_guarded using bound function ignores reference
- Issue #1754[1498] - Running with TCP parcelport causes immediate crash or freeze
- Issue #1655[1499] - Enable zip_iterator to be used with Boost traversal iterator categories
- Issue #1591[1500] - Optimize AGAS for shared memory only operation
- Issue #1401[1501] - Need an efficient infiniband parcelport
- Issue #1125[1502] - Fix the IPC parcelport
- Issue #839[1503] - Refactor ibverbs and shmem parcelport
- Issue #702[1504] - Add instrumentation of parcel layer
- Issue #668[1505] - Implement ispc task interface
- Issue #533[1506] - Thread queue/deque internal parameters should be runtime configurable
- Issue #475[1507] - Create a means of combining performance counters into querysets

1486 https://github.com/STEllAR-GROUP/hpx/pull/2245
1487 https://github.com/STEllAR-GROUP/hpx/issues/2243
1488 https://github.com/STEllAR-GROUP/hpx/issues/2242
1489 https://github.com/STEllAR-GROUP/hpx/issues/2241
1490 https://github.com/STEllAR-GROUP/hpx/pull/2240
1491 https://github.com/STEllAR-GROUP/hpx/issues/2239
1492 https://github.com/STEllAR-GROUP/hpx/issues/2236
1493 https://github.com/STEllAR-GROUP/hpx/pull/2232
1494 https://github.com/STEllAR-GROUP/hpx/issues/2147
1495 https://github.com/STEllAR-GROUP/hpx/issues/2109
1496 https://github.com/STEllAR-GROUP/hpx/issues/1973
1497 https://github.com/STEllAR-GROUP/hpx/issues/1864
1498 https://github.com/STEllAR-GROUP/hpx/issues/1754
1499 https://github.com/STEllAR-GROUP/hpx/issues/1655
1500 https://github.com/STEllAR-GROUP/hpx/issues/1591
1501 https://github.com/STEllAR-GROUP/hpx/issues/1401
1502 https://github.com/STEllAR-GROUP/hpx/issues/1125
1503 https://github.com/STEllAR-GROUP/hpx/issues/839
1504 https://github.com/STEllAR-GROUP/hpx/issues/702
1505 https://github.com/STEllAR-GROUP/hpx/issues/668
1506 https://github.com/STEllAR-GROUP/hpx/issues/533
1507 https://github.com/STEllAR-GROUP/hpx/issues/475

2.10.6 *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[1508]) proposal. This also includes the addition of projections to the existing algorithms. Please see Issue #1668[1509] for a list of algorithms which have been adapted to N4560[1510].
- We have implemented index-based parallel for-loops based on a corresponding standardization proposal (P0075R1[1511]). Please see Issue #2016[1512] 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[1513] for an overview of which algorithms are available by now.
- We have started to implement a new prototypical functionality with *HPX.Compute* which uniformly exposes some of the higher level APIs to heterogeneous architectures (currently CUDA). This functionality is an early preview and should not be considered stable. It may change considerably in the future.
- We have pervasively added (optional) executor arguments to all API functions which schedule new work. Executors are now used throughout the code base as the main means of executing tasks.
- Added `hpx::make_future<R>(future<T> &&)` allowing to convert a future of any type `T` into a future of any other type `R`, either based on default conversion rules of the embedded types or using a given explicit conversion function.
- We finally finished the implementation of transparent migration of components to another locality. It is now possible to trigger a migration operation without 'stopping the world' for the object to migrate. *HPX* will make sure that no work is being performed on an object before it is migrated and that all subsequently scheduled work for the migrated object will be transparently forwarded to the new locality. Please note that the global id of the migrated object does not change, thus the application will not have to be changed in any way to support this new functionality. Please note that this feature is currently considered experimental. See Issue #559[1514] and PR #1966[1515] for more details.

[1508] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf
[1509] https://github.com/STEllAR-GROUP/hpx/issues/1668
[1510] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf
[1511] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0075r1.pdf
[1512] https://github.com/STEllAR-GROUP/hpx/issues/2016
[1513] https://github.com/STEllAR-GROUP/hpx/issues/1141
[1514] https://github.com/STEllAR-GROUP/hpx/issues/559
[1515] https://github.com/STEllAR-GROUP/hpx/pull/1966

- 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[1516] and PR #1912[1517] 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[1518]) 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[1519] is defined at configuration time. Please explicitly qualify all uses of the dataflow facility if you enable this compatibility setting and encounter ambiguities.
- The adaptation of the C++ Extensions for Ranges (N4560[1520]) proposal imposes some breaking changes related to the return types of some of the parallel algorithms. Please see Issue #1668[1521] 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.

[1516] https://github.com/STEllAR-GROUP/hpx/issues/1265
[1517] https://github.com/STEllAR-GROUP/hpx/pull/1912
[1518] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf
[1519] https://www.cmake.org
[1520] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf
[1521] https://github.com/STEllAR-GROUP/hpx/issues/1668

- 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[1522] - change default chunker of parallel executor to static one
- PR #2247[1523] - HPX on ppc64le
- PR #2244[1524] - Fixing MSVC problems
- PR #2238[1525] - Fixing small typos
- PR #2237[1526] - Fixing small typos
- PR #2234[1527] - Fix broken add test macro when extra args are passed in
- PR #2231[1528] - Fixing possible race during future awaiting in serialization
- PR #2230[1529] - Fix stream nvcc
- PR #2229[1530] - Fixed run_as_hpx_thread
- PR #2228[1531] - On prefetching_test branch : adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2227[1532] - Support for HPXCL's opencl::event
- PR #2226[1533] - Preparing for release of V0.9.99
- PR #2225[1534] - fix issue when compiling components with hpxcxx
- PR #2224[1535] - Compute alloc fix
- PR #2223[1536] - Simplify promise
- PR #2222[1537] - Replace last uses of boost::function by util::function_nonser

[1522] https://github.com/STEllAR-GROUP/hpx/pull/2250
[1523] https://github.com/STEllAR-GROUP/hpx/pull/2247
[1524] https://github.com/STEllAR-GROUP/hpx/pull/2244
[1525] https://github.com/STEllAR-GROUP/hpx/pull/2238
[1526] https://github.com/STEllAR-GROUP/hpx/pull/2237
[1527] https://github.com/STEllAR-GROUP/hpx/pull/2234
[1528] https://github.com/STEllAR-GROUP/hpx/pull/2231
[1529] https://github.com/STEllAR-GROUP/hpx/pull/2230
[1530] https://github.com/STEllAR-GROUP/hpx/pull/2229
[1531] https://github.com/STEllAR-GROUP/hpx/pull/2228
[1532] https://github.com/STEllAR-GROUP/hpx/pull/2227
[1533] https://github.com/STEllAR-GROUP/hpx/pull/2226
[1534] https://github.com/STEllAR-GROUP/hpx/pull/2225
[1535] https://github.com/STEllAR-GROUP/hpx/pull/2224
[1536] https://github.com/STEllAR-GROUP/hpx/pull/2223
[1537] https://github.com/STEllAR-GROUP/hpx/pull/2222

- PR #2221[1538] - Fix config tests
- PR #2220[1539] - Fixing gcc 4.6 compilation issues
- PR #2219[1540] - nullptr support for `[unique_]function`
- PR #2218[1541] - Introducing clang tidy
- PR #2216[1542] - Replace NULL with nullptr
- Issue #2214[1543] - Let inspect flag use of NULL, suggest nullptr instead
- PR #2213[1544] - Require support for nullptr
- PR #2212[1545] - Properly find jemalloc through pkg-config
- PR #2211[1546] - Disable a couple of warnings reported by Intel on Windows
- PR #2210[1547] - Fixed host::block_allocator::bulk_construct
- PR #2209[1548] - Started to clean up new sort algorithms, made things compile for sort_by_key
- PR #2208[1549] - A couple of fixes that were exposed by a new sort algorithm
- PR #2207[1550] - Adding missing includes in /hpx/include/serialization.hpp
- PR #2206[1551] - Call package_action::get_future before package_action::apply
- PR #2205[1552] - The indirect_packaged_task::operator() needs to be run on a HPX thread
- PR #2204[1553] - Variadic executor parameters
- PR #2203[1554] - Delay-initialize members of partitoned iterator
- PR #2202[1555] - Added segmented fill for hpx::vector
- Issue #2201[1556] - Null Thread id encountered on partitioned_vector
- PR #2200[1557] - Fix hangs
- PR #2199[1558] - Deprecating hpx/traits.hpp
- PR #2198[1559] - Making explicit inclusion of external libraries into build
- PR #2197[1560] - Fix typo in QT CMakeLists

[1538] https://github.com/STEllAR-GROUP/hpx/pull/2221
[1539] https://github.com/STEllAR-GROUP/hpx/pull/2220
[1540] https://github.com/STEllAR-GROUP/hpx/pull/2219
[1541] https://github.com/STEllAR-GROUP/hpx/pull/2218
[1542] https://github.com/STEllAR-GROUP/hpx/pull/2216
[1543] https://github.com/STEllAR-GROUP/hpx/issues/2214
[1544] https://github.com/STEllAR-GROUP/hpx/pull/2213
[1545] https://github.com/STEllAR-GROUP/hpx/pull/2212
[1546] https://github.com/STEllAR-GROUP/hpx/pull/2211
[1547] https://github.com/STEllAR-GROUP/hpx/pull/2210
[1548] https://github.com/STEllAR-GROUP/hpx/pull/2209
[1549] https://github.com/STEllAR-GROUP/hpx/pull/2208
[1550] https://github.com/STEllAR-GROUP/hpx/pull/2207
[1551] https://github.com/STEllAR-GROUP/hpx/pull/2206
[1552] https://github.com/STEllAR-GROUP/hpx/pull/2205
[1553] https://github.com/STEllAR-GROUP/hpx/pull/2204
[1554] https://github.com/STEllAR-GROUP/hpx/pull/2203
[1555] https://github.com/STEllAR-GROUP/hpx/pull/2202
[1556] https://github.com/STEllAR-GROUP/hpx/issues/2201
[1557] https://github.com/STEllAR-GROUP/hpx/pull/2200
[1558] https://github.com/STEllAR-GROUP/hpx/pull/2199
[1559] https://github.com/STEllAR-GROUP/hpx/pull/2198
[1560] https://github.com/STEllAR-GROUP/hpx/pull/2197

- PR #2196[1561] - Fixing a gcc warning about attributes being ignored
- PR #2194[1562] - Fixing partitioned_vector_spmd_foreach example
- Issue #2193[1563] - partitioned_vector_spmd_foreach seg faults
- PR #2192[1564] - Support Boost.Thread v4
- PR #2191[1565] - HPX.Compute prototype
- PR #2190[1566] - Spawning operation on new thread if remaining stack space becomes too small
- PR #2189[1567] - Adding callback taking index and future to when_each
- PR #2188[1568] - Adding new example demonstrating receive_buffer
- PR #2187[1569] - Mask 128-bit ints if CUDA is being used
- PR #2186[1570] - Make startup & shutdown functions unique_function
- PR #2185[1571] - Fixing logging output not to cause hang on shutdown
- PR #2184[1572] - Allowing component clients as action return types
- Issue #2183[1573] - Enabling logging output causes hang on shutdown
- Issue #2182[1574] - 1d_stencil seg fault
- Issue #2181[1575] - Setting small stack size does not change default
- PR #2180[1576] - Changing default bind mode to balanced
- PR #2179[1577] - adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2177[1578] - Fixing 2176
- Issue #2176[1579] - Launch process test fails on OSX
- PR #2175[1580] - Fix unbalanced config/warnings includes, add some new ones
- PR #2174[1581] - Fix test categorization : regression not unit
- Issue #2172[1582] - Different performance results
- Issue #2171[1583] - "negative entry in reference count table" running octotiger on 32 nodes on queenbee

1561 https://github.com/STEllAR-GROUP/hpx/pull/2196
1562 https://github.com/STEllAR-GROUP/hpx/pull/2194
1563 https://github.com/STEllAR-GROUP/hpx/issues/2193
1564 https://github.com/STEllAR-GROUP/hpx/pull/2192
1565 https://github.com/STEllAR-GROUP/hpx/pull/2191
1566 https://github.com/STEllAR-GROUP/hpx/pull/2190
1567 https://github.com/STEllAR-GROUP/hpx/pull/2189
1568 https://github.com/STEllAR-GROUP/hpx/pull/2188
1569 https://github.com/STEllAR-GROUP/hpx/pull/2187
1570 https://github.com/STEllAR-GROUP/hpx/pull/2186
1571 https://github.com/STEllAR-GROUP/hpx/pull/2185
1572 https://github.com/STEllAR-GROUP/hpx/pull/2184
1573 https://github.com/STEllAR-GROUP/hpx/issues/2183
1574 https://github.com/STEllAR-GROUP/hpx/issues/2182
1575 https://github.com/STEllAR-GROUP/hpx/issues/2181
1576 https://github.com/STEllAR-GROUP/hpx/pull/2180
1577 https://github.com/STEllAR-GROUP/hpx/pull/2179
1578 https://github.com/STEllAR-GROUP/hpx/pull/2177
1579 https://github.com/STEllAR-GROUP/hpx/issues/2176
1580 https://github.com/STEllAR-GROUP/hpx/pull/2175
1581 https://github.com/STEllAR-GROUP/hpx/pull/2174
1582 https://github.com/STEllAR-GROUP/hpx/issues/2172
1583 https://github.com/STEllAR-GROUP/hpx/issues/2171

- Issue #2170[1584] - Error while compiling on Mac + boost 1.60
- PR #2168[1585] - Fixing problems with is_bitwise_serializable
- Issue #2167[1586] - startup & shutdown function should accept unique_function
- Issue #2166[1587] - Simple receive_buffer example
- PR #2165[1588] - Fix wait all
- PR #2164[1589] - Fix wait all
- PR #2163[1590] - Fix some typos in config tests
- PR #2162[1591] - Improve #includes
- PR #2160[1592] - Add inspect check for missing #include <list>
- PR #2159[1593] - Add missing finalize call to stop test hanging
- PR #2158[1594] - Algo fixes
- PR #2157[1595] - Stack check
- Issue #2156[1596] - OSX reports stack space incorrectly (generic context coroutines)
- Issue #2155[1597] - Race condition suspected in runtime
- PR #2154[1598] - Replace boost::detail::atomic_count with the new util::atomic_count
- PR #2153[1599] - Fix stack overflow on OSX
- PR #2152[1600] - Define is_bitwise_serializable as is_trivially_copyable when available
- PR #2151[1601] - Adding missing <cstring> for std::mem* functions
- Issue #2150[1602] - Unable to use component clients as action return types
- PR #2149[1603] - std::memmove copies bytes, use bytes*sizeof(type) when copying larger types
- PR #2146[1604] - Adding customization point for parallel copy/move
- PR #2145[1605] - Applying changes to address warnings issued by latest version of PVS Studio
- Issue #2148[1606] - hpx::parallel::copy is broken after trivially copyable changes

1584 https://github.com/STEllAR-GROUP/hpx/issues/2170
1585 https://github.com/STEllAR-GROUP/hpx/pull/2168
1586 https://github.com/STEllAR-GROUP/hpx/issues/2167
1587 https://github.com/STEllAR-GROUP/hpx/issues/2166
1588 https://github.com/STEllAR-GROUP/hpx/pull/2165
1589 https://github.com/STEllAR-GROUP/hpx/pull/2164
1590 https://github.com/STEllAR-GROUP/hpx/pull/2163
1591 https://github.com/STEllAR-GROUP/hpx/pull/2162
1592 https://github.com/STEllAR-GROUP/hpx/pull/2160
1593 https://github.com/STEllAR-GROUP/hpx/pull/2159
1594 https://github.com/STEllAR-GROUP/hpx/pull/2158
1595 https://github.com/STEllAR-GROUP/hpx/pull/2157
1596 https://github.com/STEllAR-GROUP/hpx/issues/2156
1597 https://github.com/STEllAR-GROUP/hpx/issues/2155
1598 https://github.com/STEllAR-GROUP/hpx/pull/2154
1599 https://github.com/STEllAR-GROUP/hpx/pull/2153
1600 https://github.com/STEllAR-GROUP/hpx/pull/2152
1601 https://github.com/STEllAR-GROUP/hpx/pull/2151
1602 https://github.com/STEllAR-GROUP/hpx/issues/2150
1603 https://github.com/STEllAR-GROUP/hpx/pull/2149
1604 https://github.com/STEllAR-GROUP/hpx/pull/2146
1605 https://github.com/STEllAR-GROUP/hpx/pull/2145
1606 https://github.com/STEllAR-GROUP/hpx/issues/2148

- PR #2144[1607] - Some minor tweaks to compute prototype
- PR #2143[1608] - Added Boost version support information over OSX platform
- PR #2142[1609] - Fixing memory leak in example
- PR #2141[1610] - Add missing specializations in execution policies
- PR #2139[1611] - This PR fixes a few problems reported by Clang's Undefined Behavior sanitizer
- PR #2138[1612] - Revert "Adding fedora docs"
- PR #2136[1613] - Removed double semicolon
- PR #2135[1614] - Add deprecated #include check for hpx_fwd.hpp
- PR #2134[1615] - Resolved memory leak in stencil_8
- PR #2133[1616] - Replace uses of boost pointer containers
- PR #2132[1617] - Removing unused typedef
- PR #2131[1618] - Add several include checks for std facilities
- PR #2130[1619] - Fixing parcel compression, adding test
- PR #2129[1620] - Fix invalid attribute warnings
- Issue #2128[1621] - hpx::init seems to segfault
- PR #2127[1622] - Making executor_traits N-nary
- PR #2126[1623] - GCC 4.6 fails to deduce the correct type in lambda
- PR #2125[1624] - Making parcel coalescing test actually test something
- Issue #2124[1625] - Make a testcase for parcel compression
- Issue #2123[1626] - hpx/hpx/runtime/applier_fwd.hpp - Multiple defined types
- Issue #2122[1627] - Exception in primary_namespace::resolve_free_list
- Issue #2121[1628] - Possible memory leak in 1d_stencil_8
- PR #2120[1629] - Fixing 2119

[1607] https://github.com/STEllAR-GROUP/hpx/pull/2144
[1608] https://github.com/STEllAR-GROUP/hpx/pull/2143
[1609] https://github.com/STEllAR-GROUP/hpx/pull/2142
[1610] https://github.com/STEllAR-GROUP/hpx/pull/2141
[1611] https://github.com/STEllAR-GROUP/hpx/pull/2139
[1612] https://github.com/STEllAR-GROUP/hpx/pull/2138
[1613] https://github.com/STEllAR-GROUP/hpx/pull/2136
[1614] https://github.com/STEllAR-GROUP/hpx/pull/2135
[1615] https://github.com/STEllAR-GROUP/hpx/pull/2134
[1616] https://github.com/STEllAR-GROUP/hpx/pull/2133
[1617] https://github.com/STEllAR-GROUP/hpx/pull/2132
[1618] https://github.com/STEllAR-GROUP/hpx/pull/2131
[1619] https://github.com/STEllAR-GROUP/hpx/pull/2130
[1620] https://github.com/STEllAR-GROUP/hpx/pull/2129
[1621] https://github.com/STEllAR-GROUP/hpx/issues/2128
[1622] https://github.com/STEllAR-GROUP/hpx/pull/2127
[1623] https://github.com/STEllAR-GROUP/hpx/pull/2126
[1624] https://github.com/STEllAR-GROUP/hpx/pull/2125
[1625] https://github.com/STEllAR-GROUP/hpx/issues/2124
[1626] https://github.com/STEllAR-GROUP/hpx/issues/2123
[1627] https://github.com/STEllAR-GROUP/hpx/issues/2122
[1628] https://github.com/STEllAR-GROUP/hpx/issues/2121
[1629] https://github.com/STEllAR-GROUP/hpx/pull/2120

- Issue #2119[1630] - reduce_by_key compilation problems
- Issue #2118[1631] - Premature unwrapping of boost::ref'ed arguments
- PR #2117[1632] - Added missing initializer on last constructor for thread_description
- PR #2116[1633] - Use a lightweight bind implementation when no placeholders are given
- PR #2115[1634] - Replace boost::shared_ptr with std::shared_ptr
- PR #2114[1635] - Adding hook functions for executor_parameter_traits supporting timers
- Issue #2113[1636] - Compilation error with gcc version 4.9.3 (MacPorts gcc49 4.9.3_0)
- PR #2112[1637] - Replace uses of safe_bool with explicit operator bool
- Issue #2111[1638] - Compilation error on QT example
- Issue #2110[1639] - Compilation error when passing non-future argument to unwrapped continuation in dataflow
- Issue #2109[1640] - Warning while compiling hpx
- Issue #2109[1641] - Stack trace of last bug causing issues with octotiger
- Issue #2108[1642] - Stack trace of last bug causing issues with octotiger
- PR #2107[1643] - Making sure that a missing parcel_coalescing module does not cause startup exceptions
- PR #2106[1644] - Stop using hpx_fwd.hpp
- Issue #2105[1645] - coalescing plugin handler is not optional any more
- Issue #2104[1646] - Make executor_traits N-nary
- Issue #2103[1647] - Build error with octotiger and hpx commit e657426d
- PR #2102[1648] - Combining thread data storage
- PR #2101[1649] - Added repartition version of 1d stencil that uses any performance counter
- PR #2100[1650] - Drop obsolete TR1 result_of protocol
- PR #2099[1651] - Replace uses of boost::bind with util::bind
- PR #2098[1652] - Deprecated inspect checks

[1630] https://github.com/STEllAR-GROUP/hpx/issues/2119
[1631] https://github.com/STEllAR-GROUP/hpx/issues/2118
[1632] https://github.com/STEllAR-GROUP/hpx/pull/2117
[1633] https://github.com/STEllAR-GROUP/hpx/pull/2116
[1634] https://github.com/STEllAR-GROUP/hpx/pull/2115
[1635] https://github.com/STEllAR-GROUP/hpx/pull/2114
[1636] https://github.com/STEllAR-GROUP/hpx/issues/2113
[1637] https://github.com/STEllAR-GROUP/hpx/pull/2112
[1638] https://github.com/STEllAR-GROUP/hpx/issues/2111
[1639] https://github.com/STEllAR-GROUP/hpx/issues/2110
[1640] https://github.com/STEllAR-GROUP/hpx/issues/2109
[1641] https://github.com/STEllAR-GROUP/hpx/issues/2109
[1642] https://github.com/STEllAR-GROUP/hpx/issues/2108
[1643] https://github.com/STEllAR-GROUP/hpx/pull/2107
[1644] https://github.com/STEllAR-GROUP/hpx/pull/2106
[1645] https://github.com/STEllAR-GROUP/hpx/issues/2105
[1646] https://github.com/STEllAR-GROUP/hpx/issues/2104
[1647] https://github.com/STEllAR-GROUP/hpx/issues/2103
[1648] https://github.com/STEllAR-GROUP/hpx/pull/2102
[1649] https://github.com/STEllAR-GROUP/hpx/pull/2101
[1650] https://github.com/STEllAR-GROUP/hpx/pull/2100
[1651] https://github.com/STEllAR-GROUP/hpx/pull/2099
[1652] https://github.com/STEllAR-GROUP/hpx/pull/2098

- PR #2097[1653] - Reduce by key, extends #1141
- PR #2096[1654] - Moving local cache from external to hpx/util
- PR #2095[1655] - Bump minimum required Boost to 1.50.0
- PR #2094[1656] - Add include checks for several Boost utilities
- Issue #2093[1657] - /.../local_cache.hpp(89): error #303: explicit type is missing ("int" assumed)
- PR #2091[1658] - Fix for Raspberry pi build
- PR #2090[1659] - Fix storage size for util::function<>
- PR #2089[1660] - Fix #2088
- Issue #2088[1661] - More verbose output from cmake configuration
- PR #2087[1662] - Making sure init_globally always executes hpx_main
- Issue #2086[1663] - Race condition with recent HPX
- PR #2085[1664] - Adding #include checker
- PR #2084[1665] - Replace boost lock types with standard library ones
- PR #2083[1666] - Simplify packaged task
- PR #2082[1667] - Updating APEX version for testing
- PR #2081[1668] - Cleanup exception headers
- PR #2080[1669] - Make call_once variadic
- Issue #2079[1670] - With GNU C++, line 85 of hpx/config/version.hpp causes link failure when linking application
- Issue #2078[1671] - Simple test fails with _GLIBCXX_DEBUG defined
- PR #2077[1672] - Instantiate board in nqueen client
- PR #2076[1673] - Moving coalescing registration to TUs
- PR #2075[1674] - Fixed some documentation typos
- PR #2074[1675] - Adding flush-mode to message handler flush

[1653] https://github.com/STEllAR-GROUP/hpx/pull/2097
[1654] https://github.com/STEllAR-GROUP/hpx/pull/2096
[1655] https://github.com/STEllAR-GROUP/hpx/pull/2095
[1656] https://github.com/STEllAR-GROUP/hpx/pull/2094
[1657] https://github.com/STEllAR-GROUP/hpx/issues/2093
[1658] https://github.com/STEllAR-GROUP/hpx/pull/2091
[1659] https://github.com/STEllAR-GROUP/hpx/pull/2090
[1660] https://github.com/STEllAR-GROUP/hpx/pull/2089
[1661] https://github.com/STEllAR-GROUP/hpx/issues/2088
[1662] https://github.com/STEllAR-GROUP/hpx/pull/2087
[1663] https://github.com/STEllAR-GROUP/hpx/issues/2086
[1664] https://github.com/STEllAR-GROUP/hpx/pull/2085
[1665] https://github.com/STEllAR-GROUP/hpx/pull/2084
[1666] https://github.com/STEllAR-GROUP/hpx/pull/2083
[1667] https://github.com/STEllAR-GROUP/hpx/pull/2082
[1668] https://github.com/STEllAR-GROUP/hpx/pull/2081
[1669] https://github.com/STEllAR-GROUP/hpx/pull/2080
[1670] https://github.com/STEllAR-GROUP/hpx/issues/2079
[1671] https://github.com/STEllAR-GROUP/hpx/issues/2078
[1672] https://github.com/STEllAR-GROUP/hpx/pull/2077
[1673] https://github.com/STEllAR-GROUP/hpx/pull/2076
[1674] https://github.com/STEllAR-GROUP/hpx/pull/2075
[1675] https://github.com/STEllAR-GROUP/hpx/pull/2074

- PR #2073[1676] - Fixing performance regression introduced lately
- PR #2072[1677] - Refactor local::condition_variable
- PR #2071[1678] - Timer based on boost::asio::deadline_timer
- PR #2070[1679] - Refactor tuple based functionality
- PR #2069[1680] - Fixed typos
- Issue #2068[1681] - Seg fault with octotiger
- PR #2067[1682] - Algorithm cleanup
- PR #2066[1683] - Split credit fixes
- PR #2065[1684] - Rename HPX_MOVABLE_BUT_NOT_COPYABLE to HPX_MOVABLE_ONLY
- PR #2064[1685] - Fixed some typos in docs
- PR #2063[1686] - Adding example demonstrating template components
- Issue #2062[1687] - Support component templates
- PR #2061[1688] - Replace some uses of lexical_cast<string> with C++11 std::to_string
- PR #2060[1689] - Replace uses of boost::noncopyable with HPX_NON_COPYABLE
- PR #2059[1690] - Adding missing for_loop algorithms
- PR #2058[1691] - Move several definitions to more appropriate headers
- PR #2057[1692] - Simplify assert_owns_lock and ignore_while_checking
- PR #2056[1693] - Replacing std::result_of with util::result_of
- PR #2055[1694] - Fix process launching/connecting back
- PR #2054[1695] - Add a forwarding coroutine header
- PR #2053[1696] - Replace uses of boost::unordered_map with std::unordered_map
- PR #2052[1697] - Rewrite tuple unwrap
- PR #2050[1698] - Replace uses of BOOST_SCOPED_ENUM with C++11 scoped enums

[1676] https://github.com/STEllAR-GROUP/hpx/pull/2073
[1677] https://github.com/STEllAR-GROUP/hpx/pull/2072
[1678] https://github.com/STEllAR-GROUP/hpx/pull/2071
[1679] https://github.com/STEllAR-GROUP/hpx/pull/2070
[1680] https://github.com/STEllAR-GROUP/hpx/pull/2069
[1681] https://github.com/STEllAR-GROUP/hpx/issues/2068
[1682] https://github.com/STEllAR-GROUP/hpx/pull/2067
[1683] https://github.com/STEllAR-GROUP/hpx/pull/2066
[1684] https://github.com/STEllAR-GROUP/hpx/pull/2065
[1685] https://github.com/STEllAR-GROUP/hpx/pull/2064
[1686] https://github.com/STEllAR-GROUP/hpx/pull/2063
[1687] https://github.com/STEllAR-GROUP/hpx/issues/2062
[1688] https://github.com/STEllAR-GROUP/hpx/pull/2061
[1689] https://github.com/STEllAR-GROUP/hpx/pull/2060
[1690] https://github.com/STEllAR-GROUP/hpx/pull/2059
[1691] https://github.com/STEllAR-GROUP/hpx/pull/2058
[1692] https://github.com/STEllAR-GROUP/hpx/pull/2057
[1693] https://github.com/STEllAR-GROUP/hpx/pull/2056
[1694] https://github.com/STEllAR-GROUP/hpx/pull/2055
[1695] https://github.com/STEllAR-GROUP/hpx/pull/2054
[1696] https://github.com/STEllAR-GROUP/hpx/pull/2053
[1697] https://github.com/STEllAR-GROUP/hpx/pull/2052
[1698] https://github.com/STEllAR-GROUP/hpx/pull/2050

- PR #2049[1699] - Attempt to narrow down split_credit problem
- PR #2048[1700] - Fixing gcc startup hangs
- PR #2047[1701] - Fixing when_xxx and wait_xxx for MSVC12
- PR #2046[1702] - adding persistent_auto_chunk_size and related tests for for_each
- PR #2045[1703] - Fixing HPX_HAVE_THREAD_BACKTRACE_DEPTH build time configuration
- PR #2044[1704] - Adding missing service executor types
- PR #2043[1705] - Removing ambiguous definitions for is_future_range and future_range_traits
- PR #2042[1706] - Clarify that HPX builds can use (much) more than 2GB per process
- PR #2041[1707] - Changing future_iterator_traits to support pointers
- Issue #2040[1708] - Improve documentation memory usage warning?
- PR #2039[1709] - Coroutine cleanup
- PR #2038[1710] - Fix cmake policy CMP0042 warning MACOSX_RPATH
- PR #2037[1711] - Avoid redundant specialization of [**unique_**]function_nonser
- PR #2036[1712] - nvcc dies with an internal error upon pushing/popping warnings inside templates
- Issue #2035[1713] - Use a less restrictive iterator definition in hpx::lcos::detail::future_iterator_traits
- PR #2034[1714] - Fixing compilation error with thread queue wait time performance counter
- Issue #2033[1715] - Compilation error when compiling with thread queue waittime performance counter
- Issue #2032[1716] - Ambiguous template instantiation for is_future_range and future_range_traits.
- PR #2031[1717] - Don't restart timer on every incoming parcel
- PR #2030[1718] - Unify handling of execution policies in parallel algorithms
- PR #2029[1719] - Make pkg-config .pc files use .dylib on OSX
- PR #2028[1720] - Adding process component
- PR #2027[1721] - Making check for compiler compatibility independent on compiler path

[1699] https://github.com/STEllAR-GROUP/hpx/pull/2049
[1700] https://github.com/STEllAR-GROUP/hpx/pull/2048
[1701] https://github.com/STEllAR-GROUP/hpx/pull/2047
[1702] https://github.com/STEllAR-GROUP/hpx/pull/2046
[1703] https://github.com/STEllAR-GROUP/hpx/pull/2045
[1704] https://github.com/STEllAR-GROUP/hpx/pull/2044
[1705] https://github.com/STEllAR-GROUP/hpx/pull/2043
[1706] https://github.com/STEllAR-GROUP/hpx/pull/2042
[1707] https://github.com/STEllAR-GROUP/hpx/pull/2041
[1708] https://github.com/STEllAR-GROUP/hpx/issues/2040
[1709] https://github.com/STEllAR-GROUP/hpx/pull/2039
[1710] https://github.com/STEllAR-GROUP/hpx/pull/2038
[1711] https://github.com/STEllAR-GROUP/hpx/pull/2037
[1712] https://github.com/STEllAR-GROUP/hpx/pull/2036
[1713] https://github.com/STEllAR-GROUP/hpx/issues/2035
[1714] https://github.com/STEllAR-GROUP/hpx/pull/2034
[1715] https://github.com/STEllAR-GROUP/hpx/issues/2033
[1716] https://github.com/STEllAR-GROUP/hpx/issues/2032
[1717] https://github.com/STEllAR-GROUP/hpx/pull/2031
[1718] https://github.com/STEllAR-GROUP/hpx/pull/2030
[1719] https://github.com/STEllAR-GROUP/hpx/pull/2029
[1720] https://github.com/STEllAR-GROUP/hpx/pull/2028
[1721] https://github.com/STEllAR-GROUP/hpx/pull/2027

- PR #2025[1722] - Fixing inspect tool
- PR #2024[1723] - Intel13 removal
- PR #2023[1724] - Fix errors related to older boost versions and parameter pack expansions in lambdas
- Issue #2022[1725] - gmake fail: "No rule to make target /usr/lib46/libboost_context-mt.so"
- PR #2021[1726] - Added Sudoku example
- Issue #2020[1727] - Make errors related to init_globally.cpp example while building HPX out of the box
- PR #2019[1728] - Fixed some compilation and cmake errors encountered in nqueen example
- PR #2018[1729] - For loop algorithms
- PR #2017[1730] - Non-recursive at_index implementation
- Issue #2016[1731] - Add index-based for-loops
- Issue #2015[1732] - Change default bind-mode to balanced
- PR #2014[1733] - Fixed dataflow if invoked action returns a future
- PR #2013[1734] - Fixing compilation issues with external example
- PR #2012[1735] - Added Sierpinski Triangle example
- Issue #2011[1736] - Compilation error while running sample hello_world_component code
- PR #2010[1737] - Segmented move implemented for hpx::vector
- Issue #2009[1738] - pkg-config order incorrect on 14.04 / GCC 4.8
- Issue #2008[1739] - Compilation error in dataflow of action returning a future
- PR #2007[1740] - Adding new performance counter exposing overall scheduler time
- PR #2006[1741] - Function includes
- PR #2005[1742] - Adding an example demonstrating how to initialize HPX from a global object
- PR #2004[1743] - Fixing 2000
- PR #2003[1744] - Adding generation parameter to gather to enable using it more than once

1722 https://github.com/STEllAR-GROUP/hpx/pull/2025
1723 https://github.com/STEllAR-GROUP/hpx/pull/2024
1724 https://github.com/STEllAR-GROUP/hpx/pull/2023
1725 https://github.com/STEllAR-GROUP/hpx/issues/2022
1726 https://github.com/STEllAR-GROUP/hpx/pull/2021
1727 https://github.com/STEllAR-GROUP/hpx/issues/2020
1728 https://github.com/STEllAR-GROUP/hpx/pull/2019
1729 https://github.com/STEllAR-GROUP/hpx/pull/2018
1730 https://github.com/STEllAR-GROUP/hpx/pull/2017
1731 https://github.com/STEllAR-GROUP/hpx/issues/2016
1732 https://github.com/STEllAR-GROUP/hpx/issues/2015
1733 https://github.com/STEllAR-GROUP/hpx/pull/2014
1734 https://github.com/STEllAR-GROUP/hpx/pull/2013
1735 https://github.com/STEllAR-GROUP/hpx/pull/2012
1736 https://github.com/STEllAR-GROUP/hpx/issues/2011
1737 https://github.com/STEllAR-GROUP/hpx/pull/2010
1738 https://github.com/STEllAR-GROUP/hpx/issues/2009
1739 https://github.com/STEllAR-GROUP/hpx/issues/2008
1740 https://github.com/STEllAR-GROUP/hpx/pull/2007
1741 https://github.com/STEllAR-GROUP/hpx/pull/2006
1742 https://github.com/STEllAR-GROUP/hpx/pull/2005
1743 https://github.com/STEllAR-GROUP/hpx/pull/2004
1744 https://github.com/STEllAR-GROUP/hpx/pull/2003

- PR #2002[1745] - Turn on position independent code to solve link problem with hpx_init
- Issue #2001[1746] - Gathering more than once segfaults
- Issue #2000[1747] - Undefined reference to hpx::assertion_failed
- Issue #1999[1748] - Seg fault in hpx::lcos::base_lco_with_value<*>::set_value_nonvirt() when running octo-tiger
- PR #1998[1749] - Detect unknown command line options
- PR #1997[1750] - Extending thread description
- PR #1996[1751] - Adding natvis files to solution (MSVC only)
- Issue #1995[1752] - Command line handling does not produce error
- PR #1994[1753] - Possible missing include in test_utils.hpp
- PR #1993[1754] - Add missing LANGUAGES tag to a hpx_add_compile_flag_if_available() call in CMakeLists.txt
- PR #1992[1755] - Fixing shared_executor_test
- PR #1991[1756] - Making sure the winsock library is properly initialized
- PR #1990[1757] - Fixing bind_test placeholder ambiguity coming from boost-1.60
- PR #1989[1758] - Performance tuning
- PR #1987[1759] - Make configurable size of internal storage in util::function
- PR #1986[1760] - AGAS Refactoring+1753 Cache mods
- PR #1985[1761] - Adding missing task_block::run() overload taking an executor
- PR #1984[1762] - Adding an optimized LRU Cache implementation (for AGAS)
- PR #1983[1763] - Avoid invoking migration table look up for all objects
- PR #1981[1764] - Replacing uintptr_t (which is not defined everywhere) with std::size_t
- PR #1980[1765] - Optimizing LCO continuations
- PR #1979[1766] - Fixing Cori
- PR #1978[1767] - Fix test check that got broken in hasty fix to memory overflow

1745 https://github.com/STEllAR-GROUP/hpx/pull/2002
1746 https://github.com/STEllAR-GROUP/hpx/issues/2001
1747 https://github.com/STEllAR-GROUP/hpx/issues/2000
1748 https://github.com/STEllAR-GROUP/hpx/issues/1999
1749 https://github.com/STEllAR-GROUP/hpx/pull/1998
1750 https://github.com/STEllAR-GROUP/hpx/pull/1997
1751 https://github.com/STEllAR-GROUP/hpx/pull/1996
1752 https://github.com/STEllAR-GROUP/hpx/issues/1995
1753 https://github.com/STEllAR-GROUP/hpx/pull/1994
1754 https://github.com/STEllAR-GROUP/hpx/pull/1993
1755 https://github.com/STEllAR-GROUP/hpx/pull/1992
1756 https://github.com/STEllAR-GROUP/hpx/pull/1991
1757 https://github.com/STEllAR-GROUP/hpx/pull/1990
1758 https://github.com/STEllAR-GROUP/hpx/pull/1989
1759 https://github.com/STEllAR-GROUP/hpx/pull/1987
1760 https://github.com/STEllAR-GROUP/hpx/pull/1986
1761 https://github.com/STEllAR-GROUP/hpx/pull/1985
1762 https://github.com/STEllAR-GROUP/hpx/pull/1984
1763 https://github.com/STEllAR-GROUP/hpx/pull/1983
1764 https://github.com/STEllAR-GROUP/hpx/pull/1981
1765 https://github.com/STEllAR-GROUP/hpx/pull/1980
1766 https://github.com/STEllAR-GROUP/hpx/pull/1979
1767 https://github.com/STEllAR-GROUP/hpx/pull/1978

- PR #1977[1768] - Refactor action traits
- PR #1976[1769] - Fixes typo in README.rst
- PR #1975[1770] - Reduce size of benchmark timing arrays to fix test failures
- PR #1974[1771] - Add action to update data owned by the partitioned_vector component
- PR #1972[1772] - Adding partitioned_vector SPMD example
- PR #1971[1773] - Fixing 1965
- PR #1970[1774] - Papi fixes
- PR #1969[1775] - Fixing continuation recursions to not depend on fixed amount of recursions
- PR #1968[1776] - More segmented algorithms
- Issue #1967[1777] - Simplify component implementations
- PR #1966[1778] - Migrate components
- Issue #1964[1779] - fatal error: 'boost/lockfree/detail/branch_hints.hpp' file not found
- Issue #1962[1780] - parallel:copy_if has race condition when used on in place arrays
- PR #1963[1781] - Fixing Static Parcelport initialization
- PR #1961[1782] - Fix function target
- Issue #1960[1783] - Papi counters don't reset
- PR #1959[1784] - Fixing 1958
- Issue #1958[1785] - inclusive_scan gives incorrect results with non-commutative operator
- PR #1957[1786] - Fixing #1950
- PR #1956[1787] - Sort by key example
- PR #1955[1788] - Adding regression test for #1946: Hang in wait_all() in distributed run
- Issue #1954[1789] - HPX releases should not use -Werror
- PR #1953[1790] - Adding performance analysis for AGAS cache

[1768] https://github.com/STEllAR-GROUP/hpx/pull/1977
[1769] https://github.com/STEllAR-GROUP/hpx/pull/1976
[1770] https://github.com/STEllAR-GROUP/hpx/pull/1975
[1771] https://github.com/STEllAR-GROUP/hpx/pull/1974
[1772] https://github.com/STEllAR-GROUP/hpx/pull/1972
[1773] https://github.com/STEllAR-GROUP/hpx/pull/1971
[1774] https://github.com/STEllAR-GROUP/hpx/pull/1970
[1775] https://github.com/STEllAR-GROUP/hpx/pull/1969
[1776] https://github.com/STEllAR-GROUP/hpx/pull/1968
[1777] https://github.com/STEllAR-GROUP/hpx/issues/1967
[1778] https://github.com/STEllAR-GROUP/hpx/pull/1966
[1779] https://github.com/STEllAR-GROUP/hpx/issues/1964
[1780] https://github.com/STEllAR-GROUP/hpx/issues/1962
[1781] https://github.com/STEllAR-GROUP/hpx/pull/1963
[1782] https://github.com/STEllAR-GROUP/hpx/pull/1961
[1783] https://github.com/STEllAR-GROUP/hpx/issues/1960
[1784] https://github.com/STEllAR-GROUP/hpx/pull/1959
[1785] https://github.com/STEllAR-GROUP/hpx/issues/1958
[1786] https://github.com/STEllAR-GROUP/hpx/pull/1957
[1787] https://github.com/STEllAR-GROUP/hpx/pull/1956
[1788] https://github.com/STEllAR-GROUP/hpx/pull/1955
[1789] https://github.com/STEllAR-GROUP/hpx/issues/1954
[1790] https://github.com/STEllAR-GROUP/hpx/pull/1953

- PR #1952[1791] - Adapting test for explicit variadics to fail for gcc 4.6
- PR #1951[1792] - Fixing memory leak
- Issue #1950[1793] - Simplify external builds
- PR #1949[1794] - Fixing yet another lock that is being held during suspension
- PR #1948[1795] - Fixed container algorithms for Intel
- PR #1947[1796] - Adding workaround for tagged_tuple
- Issue #1946[1797] - Hang in wait_all() in distributed run
- PR #1945[1798] - Fixed container algorithm tests
- Issue #1944[1799] - assertion 'p.destination_locality() == hpx::get_locality()' failed
- PR #1943[1800] - Fix a couple of compile errors with clang
- PR #1942[1801] - Making parcel coalescing functional
- Issue #1941[1802] - Re-enable parcel coalescing
- PR #1940[1803] - Touching up make_future
- PR #1939[1804] - Fixing problems in over-subscription management in the resource manager
- PR #1938[1805] - Removing use of unified Boost.Thread header
- PR #1937[1806] - Cleaning up the use of Boost.Accumulator headers
- PR #1936[1807] - Making sure interval timer is started for aggregating performance counters
- PR #1935[1808] - Tagged results
- PR #1934[1809] - Fix remote async with deferred launch policy
- Issue #1933[1810] - Floating point exception in `statistics_counter<boost::accumulators::tag::mean>::get_c`
- PR #1932[1811] - Removing superfluous includes of boost/lockfree/detail/branch_hints.hpp
- PR #1931[1812] - fix compilation with clang 3.8.0
- Issue #1930[1813] - Missing online documentation for HPX 0.9.11

[1791] https://github.com/STEllAR-GROUP/hpx/pull/1952
[1792] https://github.com/STEllAR-GROUP/hpx/pull/1951
[1793] https://github.com/STEllAR-GROUP/hpx/issues/1950
[1794] https://github.com/STEllAR-GROUP/hpx/pull/1949
[1795] https://github.com/STEllAR-GROUP/hpx/pull/1948
[1796] https://github.com/STEllAR-GROUP/hpx/pull/1947
[1797] https://github.com/STEllAR-GROUP/hpx/issues/1946
[1798] https://github.com/STEllAR-GROUP/hpx/pull/1945
[1799] https://github.com/STEllAR-GROUP/hpx/issues/1944
[1800] https://github.com/STEllAR-GROUP/hpx/pull/1943
[1801] https://github.com/STEllAR-GROUP/hpx/pull/1942
[1802] https://github.com/STEllAR-GROUP/hpx/issues/1941
[1803] https://github.com/STEllAR-GROUP/hpx/pull/1940
[1804] https://github.com/STEllAR-GROUP/hpx/pull/1939
[1805] https://github.com/STEllAR-GROUP/hpx/pull/1938
[1806] https://github.com/STEllAR-GROUP/hpx/pull/1937
[1807] https://github.com/STEllAR-GROUP/hpx/pull/1936
[1808] https://github.com/STEllAR-GROUP/hpx/pull/1935
[1809] https://github.com/STEllAR-GROUP/hpx/pull/1934
[1810] https://github.com/STEllAR-GROUP/hpx/issues/1933
[1811] https://github.com/STEllAR-GROUP/hpx/pull/1932
[1812] https://github.com/STEllAR-GROUP/hpx/pull/1931
[1813] https://github.com/STEllAR-GROUP/hpx/issues/1930

- PR #1929[1814] - LWG2485: get() should be overloaded for const tuple&&
- PR #1928[1815] - Revert "Using ninja for circle-ci builds"
- PR #1927[1816] - Using ninja for circle-ci builds
- PR #1926[1817] - Fixing serialization of std::array
- Issue #1925[1818] - Issues with static HPX libraries
- Issue #1924[1819] - Peformance degrading over time
- Issue #1923[1820] - serialization of std::array appears broken in latest commit
- PR #1922[1821] - Container algorithms
- PR #1921[1822] - Tons of smaller quality improvements
- Issue #1920[1823] - Seg fault in hpx::serialization::output_archive::add_gid when running octotiger
- Issue #1919[1824] - Intel 15 compiler bug preventing HPX build
- PR #1918[1825] - Address sanitizer fixes
- PR #1917[1826] - Fixing compilation problems of parallel::sort with Intel compilers
- PR #1916[1827] - Making sure code compiles if HPX_WITH_HWLOC=Off
- Issue #1915[1828] - max_cores undefined if HPX_WITH_HWLOC=Off
- PR #1913[1829] - Add utility member functions for partitioned_vector
- PR #1912[1830] - Adding support for invoking actions to dataflow
- PR #1911[1831] - Adding first batch of container algorithms
- PR #1910[1832] - Keep cmake_module_path
- PR #1909[1833] - Fix mpirun with pbs
- PR #1908[1834] - Changing parallel::sort to return the last iterator as proposed by N4560
- PR #1907[1835] - Adding a minimum version for Open MPI
- PR #1906[1836] - Updates to the Release Procedure

[1814] https://github.com/STEllAR-GROUP/hpx/pull/1929
[1815] https://github.com/STEllAR-GROUP/hpx/pull/1928
[1816] https://github.com/STEllAR-GROUP/hpx/pull/1927
[1817] https://github.com/STEllAR-GROUP/hpx/pull/1926
[1818] https://github.com/STEllAR-GROUP/hpx/issues/1925
[1819] https://github.com/STEllAR-GROUP/hpx/issues/1924
[1820] https://github.com/STEllAR-GROUP/hpx/issues/1923
[1821] https://github.com/STEllAR-GROUP/hpx/pull/1922
[1822] https://github.com/STEllAR-GROUP/hpx/pull/1921
[1823] https://github.com/STEllAR-GROUP/hpx/issues/1920
[1824] https://github.com/STEllAR-GROUP/hpx/issues/1919
[1825] https://github.com/STEllAR-GROUP/hpx/pull/1918
[1826] https://github.com/STEllAR-GROUP/hpx/pull/1917
[1827] https://github.com/STEllAR-GROUP/hpx/pull/1916
[1828] https://github.com/STEllAR-GROUP/hpx/issues/1915
[1829] https://github.com/STEllAR-GROUP/hpx/pull/1913
[1830] https://github.com/STEllAR-GROUP/hpx/pull/1912
[1831] https://github.com/STEllAR-GROUP/hpx/pull/1911
[1832] https://github.com/STEllAR-GROUP/hpx/pull/1910
[1833] https://github.com/STEllAR-GROUP/hpx/pull/1909
[1834] https://github.com/STEllAR-GROUP/hpx/pull/1908
[1835] https://github.com/STEllAR-GROUP/hpx/pull/1907
[1836] https://github.com/STEllAR-GROUP/hpx/pull/1906

- PR #1905[1837] - Fixing #1903
- PR #1904[1838] - Making sure std containers are cleared before serialization loads data
- Issue #1903[1839] - When running octotiger, I get: assertion `'(*new_gids_)[gid].size() == 1' failed: HPX(assertion_failure)`
- Issue #1902[1840] - Immediate crash when running hpx/octotiger with _GLIBCXX_DEBUG defined.
- PR #1901[1841] - Making non-serializable classes non-serializable
- Issue #1900[1842] - Two possible issues with std::list serialization
- PR #1899[1843] - Fixing a problem with credit splitting as revealed by #1898
- Issue #1898[1844] - Accessing component from locality where it was not created segfaults
- PR #1897[1845] - Changing parallel::sort to return the last iterator as proposed by N4560
- Issue #1896[1846] - version 1.0?
- Issue #1895[1847] - Warning comment on numa_allocator is not very clear
- PR #1894[1848] - Add support for compilers that have thread_local
- PR #1893[1849] - Fixing 1890
- PR #1892[1850] - Adds typed future_type for executor_traits
- PR #1891[1851] - Fix wording in certain parallel algorithm docs
- Issue #1890[1852] - Invoking papi counters give segfault
- PR #1889[1853] - Fixing problems as reported by clang-check
- PR #1888[1854] - WIP parallel is_heap
- PR #1887[1855] - Fixed resetting performance counters related to idle-rate, etc
- Issue #1886[1856] - Run hpx with qsub does not work
- PR #1885[1857] - Warning cleaning pass
- PR #1884[1858] - Add missing parallel algorithm header
- PR #1883[1859] - Add feature test for thread_local on Clang for TLS

[1837] https://github.com/STEllAR-GROUP/hpx/pull/1905
[1838] https://github.com/STEllAR-GROUP/hpx/pull/1904
[1839] https://github.com/STEllAR-GROUP/hpx/issues/1903
[1840] https://github.com/STEllAR-GROUP/hpx/issues/1902
[1841] https://github.com/STEllAR-GROUP/hpx/pull/1901
[1842] https://github.com/STEllAR-GROUP/hpx/issues/1900
[1843] https://github.com/STEllAR-GROUP/hpx/pull/1899
[1844] https://github.com/STEllAR-GROUP/hpx/issues/1898
[1845] https://github.com/STEllAR-GROUP/hpx/pull/1897
[1846] https://github.com/STEllAR-GROUP/hpx/issues/1896
[1847] https://github.com/STEllAR-GROUP/hpx/issues/1895
[1848] https://github.com/STEllAR-GROUP/hpx/pull/1894
[1849] https://github.com/STEllAR-GROUP/hpx/pull/1893
[1850] https://github.com/STEllAR-GROUP/hpx/pull/1892
[1851] https://github.com/STEllAR-GROUP/hpx/pull/1891
[1852] https://github.com/STEllAR-GROUP/hpx/issues/1890
[1853] https://github.com/STEllAR-GROUP/hpx/pull/1889
[1854] https://github.com/STEllAR-GROUP/hpx/pull/1888
[1855] https://github.com/STEllAR-GROUP/hpx/pull/1887
[1856] https://github.com/STEllAR-GROUP/hpx/issues/1886
[1857] https://github.com/STEllAR-GROUP/hpx/pull/1885
[1858] https://github.com/STEllAR-GROUP/hpx/pull/1884
[1859] https://github.com/STEllAR-GROUP/hpx/pull/1883

- PR #1882[1860] - Fix some redundant qualifiers
- Issue #1881[1861] - Unable to compile Octotiger using HPX and Intel MPI on SuperMIC
- Issue #1880[1862] - clang with libc++ on Linux needs TLS case
- PR #1879[1863] - Doc fixes for #1868
- PR #1878[1864] - Simplify functions
- PR #1877[1865] - Removing most usage of Boost.Config
- PR #1876[1866] - Add missing parallel algorithms to algorithm.hpp
- PR #1875[1867] - Simplify callables
- PR #1874[1868] - Address long standing FIXME on using `std::unique_ptr` with incomplete types
- PR #1873[1869] - Fixing 1871
- PR #1872[1870] - Making sure PBS environment uses specified node list even if no PBS_NODEFILE env is available
- Issue #1871[1871] - Fortran checks should be optional
- PR #1870[1872] - Touch local::mutex
- PR #1869[1873] - Documentation refactoring based off #1868
- PR #1867[1874] - Embrace static_assert
- PR #1866[1875] - Fix #1803 with documentation refactoring
- PR #1865[1876] - Setting OUTPUT_NAME as target properties
- PR #1863[1877] - Use SYSTEM for boost includes
- PR #1862[1878] - Minor cleanups
- PR #1861[1879] - Minor Corrections for Release
- PR #1860[1880] - Fixing hpx gdb script
- Issue #1859[1881] - reset_active_counters resets times and thread counts before some of the counters are evaluated
- PR #1858[1882] - Release V0.9.11

[1860] https://github.com/STEllAR-GROUP/hpx/pull/1882
[1861] https://github.com/STEllAR-GROUP/hpx/issues/1881
[1862] https://github.com/STEllAR-GROUP/hpx/issues/1880
[1863] https://github.com/STEllAR-GROUP/hpx/pull/1879
[1864] https://github.com/STEllAR-GROUP/hpx/pull/1878
[1865] https://github.com/STEllAR-GROUP/hpx/pull/1877
[1866] https://github.com/STEllAR-GROUP/hpx/pull/1876
[1867] https://github.com/STEllAR-GROUP/hpx/pull/1875
[1868] https://github.com/STEllAR-GROUP/hpx/pull/1874
[1869] https://github.com/STEllAR-GROUP/hpx/pull/1873
[1870] https://github.com/STEllAR-GROUP/hpx/pull/1872
[1871] https://github.com/STEllAR-GROUP/hpx/issues/1871
[1872] https://github.com/STEllAR-GROUP/hpx/pull/1870
[1873] https://github.com/STEllAR-GROUP/hpx/pull/1869
[1874] https://github.com/STEllAR-GROUP/hpx/pull/1867
[1875] https://github.com/STEllAR-GROUP/hpx/pull/1866
[1876] https://github.com/STEllAR-GROUP/hpx/pull/1865
[1877] https://github.com/STEllAR-GROUP/hpx/pull/1863
[1878] https://github.com/STEllAR-GROUP/hpx/pull/1862
[1879] https://github.com/STEllAR-GROUP/hpx/pull/1861
[1880] https://github.com/STEllAR-GROUP/hpx/pull/1860
[1881] https://github.com/STEllAR-GROUP/hpx/issues/1859
[1882] https://github.com/STEllAR-GROUP/hpx/pull/1858

- PR #1857[1883] - removing diskperf example from 9.11 release
- PR #1856[1884] - fix return in packaged_task_base::reset()
- Issue #1842[1885] - Install error: file INSTALL cannot find libhpx_parcel_coalescing.so.0.9.11
- PR #1839[1886] - Adding fedora docs
- PR #1824[1887] - Changing version on master to V0.9.12
- PR #1818[1888] - Fixing #1748
- Issue #1815[1889] - seg fault in AGAS
- Issue #1803[1890] - wait_all documentation
- Issue #1796[1891] - Outdated documentation to be revised
- Issue #1759[1892] - glibc munmap_chunk or free(): invalid pointer on SuperMIC
- Issue #1753[1893] - HPX performance degrades with time since execution begins
- Issue #1748[1894] - All public HPX headers need to be self contained
- PR #1719[1895] - How to build HPX with Visual Studio
- Issue #1684[1896] - Race condition when using –hpx:connect?
- PR #1658[1897] - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1641[1898] - Generic client
- Issue #1632[1899] - heartbeat example fails on separate nodes
- PR #1603[1900] - Adds preferred namespace check to inspect tool
- Issue #1559[1901] - Extend inspect tool
- Issue #1523[1902] - Remote async with deferred launch policy never executes
- Issue #1472[1903] - Serialization issues
- Issue #1457[1904] - Implement N4392: C++ Latches and Barriers
- PR #1444[1905] - Enabling usage of moveonly types for component construction

[1883] https://github.com/STEllAR-GROUP/hpx/pull/1857
[1884] https://github.com/STEllAR-GROUP/hpx/pull/1856
[1885] https://github.com/STEllAR-GROUP/hpx/issues/1842
[1886] https://github.com/STEllAR-GROUP/hpx/pull/1839
[1887] https://github.com/STEllAR-GROUP/hpx/pull/1824
[1888] https://github.com/STEllAR-GROUP/hpx/pull/1818
[1889] https://github.com/STEllAR-GROUP/hpx/issues/1815
[1890] https://github.com/STEllAR-GROUP/hpx/issues/1803
[1891] https://github.com/STEllAR-GROUP/hpx/issues/1796
[1892] https://github.com/STEllAR-GROUP/hpx/issues/1759
[1893] https://github.com/STEllAR-GROUP/hpx/issues/1753
[1894] https://github.com/STEllAR-GROUP/hpx/issues/1748
[1895] https://github.com/STEllAR-GROUP/hpx/pull/1719
[1896] https://github.com/STEllAR-GROUP/hpx/issues/1684
[1897] https://github.com/STEllAR-GROUP/hpx/pull/1658
[1898] https://github.com/STEllAR-GROUP/hpx/pull/1641
[1899] https://github.com/STEllAR-GROUP/hpx/issues/1632
[1900] https://github.com/STEllAR-GROUP/hpx/pull/1603
[1901] https://github.com/STEllAR-GROUP/hpx/issues/1559
[1902] https://github.com/STEllAR-GROUP/hpx/issues/1523
[1903] https://github.com/STEllAR-GROUP/hpx/issues/1472
[1904] https://github.com/STEllAR-GROUP/hpx/issues/1457
[1905] https://github.com/STEllAR-GROUP/hpx/pull/1444

- Issue #1407[1906] - The Intel 13 compiler has failing unit tests
- Issue #1405[1907] - Allow component constructors to take movable only types
- Issue #1265[1908] - Enable dataflow() to be usable with actions
- Issue #1236[1909] - NUMA aware allocators
- Issue #802[1910] - Fix Broken Examples
- Issue #559[1911] - Add hpx::migrate facility
- Issue #449[1912] - Make actions with template arguments usable and add documentation
- Issue #279[1913] - Refactor addressing_service into a base class and two derived classes
- Issue #224[1914] - Changing thread state metadata is not thread safe
- Issue #55[1915] - Uniform syntax for enums should be implemented

2.10.7 *HPX* V0.9.11 (Nov 11, 2015)

Our main focus for this release was the design and development of a coherent set of higher-level APIs exposing various types of parallelism to the application programmer. We introduced the concepts of an `executor`, which can be used to customize the `where` and `when` of execution of tasks in the context of parallelizing codes. We extended all APIs related to managing parallel tasks to support executors which gives the user the choce of either using one of the predefined executor types or to provide its own, possibly application specific, executor. We paid very close attention to align all of these changes with the existing C++ Standards documents or with the ongoing proposals for standardization.

This release is the first after our change to a new development policy. We switched all development to be strictly performed on branches only, all direct commits to our main branch (`master`) are prohibited. Any change has to go through a peer review before it will be merged to `master`. As a result the overall stability of our code base has significantly increased, the development process itself has been simplified. This change manifests itself in a large number of pull-requests which have been merged (please see below for a full list of closed issues and pull-requests). All in all for this release, we closed almost 100 issues and merged over 290 pull-requests. There have been over 1600 commits to the master branch since the last release.

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[1916] hosted continuous integration service. This gives us constant and immediate feedback on the health of our master branch.

[1906] https://github.com/STEllAR-GROUP/hpx/issues/1407
[1907] https://github.com/STEllAR-GROUP/hpx/issues/1405
[1908] https://github.com/STEllAR-GROUP/hpx/issues/1265
[1909] https://github.com/STEllAR-GROUP/hpx/issues/1236
[1910] https://github.com/STEllAR-GROUP/hpx/issues/802
[1911] https://github.com/STEllAR-GROUP/hpx/issues/559
[1912] https://github.com/STEllAR-GROUP/hpx/issues/449
[1913] https://github.com/STEllAR-GROUP/hpx/issues/279
[1914] https://github.com/STEllAR-GROUP/hpx/issues/224
[1915] https://github.com/STEllAR-GROUP/hpx/issues/55
[1916] https://circleci.com/gh/STEllAR-GROUP/hpx

- 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 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 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[1917].
- 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[1918]).
- Added performance counters exposing data related to data transferred by input/output (filesystem) operations (thanks to Maciej Brodowicz).
- Added performance counters allowing to track the number of action invocations (local and remote invocations).
- Added new command line options –hpx:print-counter-at and –hpx:reset-counters.
- The `hpx::vector` component has been renamed to `hpx::partitioned_vector` to make it explicit that the underlying memory is not contiguous.
- Introduced a completely new and uniform higher-level parallelism API which is based on executors. All existing parallelism APIs have been adapted to this. We have added a large number of different executor types, such as a numa-aware executor, a this-thread executor, etc.
- Added support for the MingW toolchain on Windows (thanks to Eric Lemanissier).
- HPX now includes support for APEX, (Autonomic Performance Environment for eXascale). APEX is an instrumentation and software adaptation library that provides an interface to TAU profiling / tracing as well

[1917] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
[1918] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html

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[1919] 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 compatibilty with naming conventions and other CMake projects. Generated config files go into <prefix>/lib/cmake/HPX and not <prefix>/lib/cmake/hpx.
- The macro `HPX_REGISTER_MINIMAL_COMPONENT_FACTORY` has been deprecated. Please use *`HPX_REGISTER_COMPONENT`*. instead. The old macro will be removed in the next release.
- The obsolete distributing_factory and binpacking_factory components have been removed. The corresponding functionality is now provided by the `hpx::new_` API function in conjunction with the `hpx::default_layout` and `hpx::binpacking` distribution policies (*`hpx::components::default_distribution_policy`* and *`hpx::components::binpacking_distribution_policy`*)
- The API function `hpx::new_colocated` has been deprecated. Please use the consolidated API `hpx::new_` in conjunction with the new `hpx::colocated` distribution policy (*`hpx::components::colocating_distribution_policy`*) instead. The old API function will still be available for at least one release of *HPX* if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The API function `hpx::async_colocated` has been deprecated. Please use the consolidated API `hpx::async` in conjunction with the new `hpx::colocated` distribution policy (*`hpx::components::colocating_distribution_policy`*) instead. The old API function will still be available for at least one release of *HPX* if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The obsolete remote_object component has been removed.
- Replaced the use of Boost.Serialization with our own solution. While the new version is mostly compatible with Boost.Serialization, this change requires some minor code modifications in user code. For more information, please see the corresponding announcement[1920] 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*).

[1919] https://github.com/STEllAR-GROUP/hpx/issues/1141
[1920] http://thread.gmane.org/gmane.comp.lib.hpx.devel/196

- 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[1921] - Completely removing external/endian
- PR #1854[1922] - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1853[1923] - Updating CMake configuration to get correct version of TAU library
- PR #1852[1924] - Fixing Performance Problems with MPI Parcelport
- PR #1851[1925] - Fixing hpx_add_link_flag() and hpx_remove_link_flag()
- PR #1850[1926] - Fixing 1836, adding parallel::sort
- PR #1849[1927] - Fixing configuration for use of more than 64 cores
- PR #1848[1928] - Change default APEX version for release
- PR #1847[1929] - Fix client_base::then on release
- PR #1846[1930] - Removing broken lcos::local::channel from release
- PR #1845[1931] - Adding example demonstrating a possible safe-object implementation to release
- PR #1844[1932] - Removing stubs from accumulator examples
- PR #1843[1933] - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1841[1934] - Fixing client_base<>::then
- PR #1840[1935] - Adding example demonstrating a possible safe-object implementation
- PR #1838[1936] - Update version rc1
- PR #1837[1937] - Removing broken lcos::local::channel

1921 https://github.com/STEllAR-GROUP/hpx/pull/1855
1922 https://github.com/STEllAR-GROUP/hpx/pull/1854
1923 https://github.com/STEllAR-GROUP/hpx/pull/1853
1924 https://github.com/STEllAR-GROUP/hpx/pull/1852
1925 https://github.com/STEllAR-GROUP/hpx/pull/1851
1926 https://github.com/STEllAR-GROUP/hpx/pull/1850
1927 https://github.com/STEllAR-GROUP/hpx/pull/1849
1928 https://github.com/STEllAR-GROUP/hpx/pull/1848
1929 https://github.com/STEllAR-GROUP/hpx/pull/1847
1930 https://github.com/STEllAR-GROUP/hpx/pull/1846
1931 https://github.com/STEllAR-GROUP/hpx/pull/1845
1932 https://github.com/STEllAR-GROUP/hpx/pull/1844
1933 https://github.com/STEllAR-GROUP/hpx/pull/1843
1934 https://github.com/STEllAR-GROUP/hpx/pull/1841
1935 https://github.com/STEllAR-GROUP/hpx/pull/1840
1936 https://github.com/STEllAR-GROUP/hpx/pull/1838
1937 https://github.com/STEllAR-GROUP/hpx/pull/1837

- PR #1835[1938] - Adding exlicit move constructor and assignment operator to hpx::lcos::promise
- PR #1834[1939] - Making hpx::lcos::promise move-only
- PR #1833[1940] - Adding fedora docs
- Issue #1832[1941] - hpx::lcos::promise<> must be move-only
- PR #1831[1942] - Fixing resource manager gcc5.2
- PR #1830[1943] - Fix intel13
- PR #1829[1944] - Unbreaking thread test
- PR #1828[1945] - Fixing #1620
- PR #1827[1946] - Fixing a memory management issue for the Parquet application
- Issue #1826[1947] - Memory management issue in hpx::lcos::promise
- PR #1825[1948] - Adding hpx::components::component and hpx::components::component_base
- PR #1823[1949] - Adding git commit id to circleci build
- PR #1822[1950] - applying fixes suggested by clang 3.7
- PR #1821[1951] - Hyperlink fixes
- PR #1820[1952] - added parallel multi-locality sanity test
- PR #1819[1953] - Fixing #1667
- Issue #1817[1954] - Hyperlinks generated by inspect tool are wrong
- PR #1816[1955] - Support hpxrx
- PR #1814[1956] - Fix async to dispatch to the correct locality in all cases
- Issue #1813[1957] - async(launch::. . . , action(), . . .) always invokes locally
- PR #1812[1958] - fixed syntax error in CMakeLists.txt
- PR #1811[1959] - Agas optimizations
- PR #1810[1960] - drop superfluous typedefs

[1938] https://github.com/STEllAR-GROUP/hpx/pull/1835
[1939] https://github.com/STEllAR-GROUP/hpx/pull/1834
[1940] https://github.com/STEllAR-GROUP/hpx/pull/1833
[1941] https://github.com/STEllAR-GROUP/hpx/issues/1832
[1942] https://github.com/STEllAR-GROUP/hpx/pull/1831
[1943] https://github.com/STEllAR-GROUP/hpx/pull/1830
[1944] https://github.com/STEllAR-GROUP/hpx/pull/1829
[1945] https://github.com/STEllAR-GROUP/hpx/pull/1828
[1946] https://github.com/STEllAR-GROUP/hpx/pull/1827
[1947] https://github.com/STEllAR-GROUP/hpx/issues/1826
[1948] https://github.com/STEllAR-GROUP/hpx/pull/1825
[1949] https://github.com/STEllAR-GROUP/hpx/pull/1823
[1950] https://github.com/STEllAR-GROUP/hpx/pull/1822
[1951] https://github.com/STEllAR-GROUP/hpx/pull/1821
[1952] https://github.com/STEllAR-GROUP/hpx/pull/1820
[1953] https://github.com/STEllAR-GROUP/hpx/pull/1819
[1954] https://github.com/STEllAR-GROUP/hpx/issues/1817
[1955] https://github.com/STEllAR-GROUP/hpx/pull/1816
[1956] https://github.com/STEllAR-GROUP/hpx/pull/1814
[1957] https://github.com/STEllAR-GROUP/hpx/issues/1813
[1958] https://github.com/STEllAR-GROUP/hpx/pull/1812
[1959] https://github.com/STEllAR-GROUP/hpx/pull/1811
[1960] https://github.com/STEllAR-GROUP/hpx/pull/1810

- PR #1809[1961] - Allow HPX to be used as an optional package in 3rd party code
- PR #1808[1962] - Fixing #1723
- PR #1807[1963] - Making sure resolve_localities does not hang during normal operation
- Issue #1806[1964] - Spinlock no longer movable and deletes operator '=', breaks MiniGhost
- Issue #1804[1965] - register_with_basename causes hangs
- PR #1801[1966] - Enhanced the inspect tool to take user directly to the problem with hyperlinks
- Issue #1800[1967] - Problems compiling application on smic
- PR #1799[1968] - Fixing cv exceptions
- PR #1798[1969] - Documentation refactoring & updating
- PR #1797[1970] - Updating the activeharmony CMake module
- PR #1795[1971] - Fixing cv
- PR #1794[1972] - Fix connect with hpx::runtime_mode_connect
- PR #1793[1973] - fix a wrong use of HPX_MAX_CPU_COUNT instead of HPX_HAVE_MAX_CPU_COUNT
- PR #1792[1974] - Allow for default constructed parcel instances to be moved
- PR #1791[1975] - Fix connect with hpx::runtime_mode_connect
- Issue #1790[1976] - assertion `action_.get()` failed: HPX(assertion_failure) when running Octotiger with pull request 1786
- PR #1789[1977] - Fixing discover_counter_types API function
- Issue #1788[1978] - connect with hpx::runtime_mode_connect
- Issue #1787[1979] - discover_counter_types not working
- PR #1786[1980] - Changing addressing_service to use std::unordered_map instead of std::map
- PR #1785[1981] - Fix is_iterator for container algorithms
- PR #1784[1982] - Adding new command line options:
- PR #1783[1983] - Minor changes for APEX support

1961 https://github.com/STEllAR-GROUP/hpx/pull/1809
1962 https://github.com/STEllAR-GROUP/hpx/pull/1808
1963 https://github.com/STEllAR-GROUP/hpx/pull/1807
1964 https://github.com/STEllAR-GROUP/hpx/issues/1806
1965 https://github.com/STEllAR-GROUP/hpx/issues/1804
1966 https://github.com/STEllAR-GROUP/hpx/pull/1801
1967 https://github.com/STEllAR-GROUP/hpx/issues/1800
1968 https://github.com/STEllAR-GROUP/hpx/pull/1799
1969 https://github.com/STEllAR-GROUP/hpx/pull/1798
1970 https://github.com/STEllAR-GROUP/hpx/pull/1797
1971 https://github.com/STEllAR-GROUP/hpx/pull/1795
1972 https://github.com/STEllAR-GROUP/hpx/pull/1794
1973 https://github.com/STEllAR-GROUP/hpx/pull/1793
1974 https://github.com/STEllAR-GROUP/hpx/pull/1792
1975 https://github.com/STEllAR-GROUP/hpx/pull/1791
1976 https://github.com/STEllAR-GROUP/hpx/issues/1790
1977 https://github.com/STEllAR-GROUP/hpx/pull/1789
1978 https://github.com/STEllAR-GROUP/hpx/issues/1788
1979 https://github.com/STEllAR-GROUP/hpx/issues/1787
1980 https://github.com/STEllAR-GROUP/hpx/pull/1786
1981 https://github.com/STEllAR-GROUP/hpx/pull/1785
1982 https://github.com/STEllAR-GROUP/hpx/pull/1784
1983 https://github.com/STEllAR-GROUP/hpx/pull/1783

- PR #1782[1984] - Drop legacy forwarding action traits
- PR #1781[1985] - Attempt to resolve the race between cv::wait_xxx and cv::notify_all
- PR #1780[1986] - Removing serialize_sequence
- PR #1779[1987] - Fixed #1501: hwloc configuration options are wrong for MIC
- PR #1778[1988] - Removing ability to enable/disable parcel handling
- PR #1777[1989] - Completely removing stackless threads
- PR #1776[1990] - Cleaning up util/plugin
- PR #1775[1991] - Agas fixes
- PR #1774[1992] - Action invocation count
- PR #1773[1993] - replaced MSVC variable with WIN32
- PR #1772[1994] - Fixing Problems in MPI parcelport and future serialization.
- PR #1771[1995] - Fixing intel 13 compiler errors related to variadic template template parameters for `lcos::when_` tests
- PR #1770[1996] - Forwarding decay to `std::`
- PR #1769[1997] - Add more characters with special regex meaning to the existing patch
- PR #1768[1998] - Adding test for receive_buffer
- PR #1767[1999] - Making sure that uptime counter throws exception on any attempt to be reset
- PR #1766[2000] - Cleaning up code related to throttling scheduler
- PR #1765[2001] - Restricting thread_data to creating only with intrusive_pointers
- PR #1764[2002] - Fixing 1763
- Issue #1763[2003] - UB in thread_data::operator delete
- PR #1762[2004] - Making sure all serialization registries/factories are unique
- PR #1761[2005] - Fixed #1751: hpx::future::wait_for fails a simple test
- PR #1758[2006] - Fixing #1757

1984 https://github.com/STEllAR-GROUP/hpx/pull/1782
1985 https://github.com/STEllAR-GROUP/hpx/pull/1781
1986 https://github.com/STEllAR-GROUP/hpx/pull/1780
1987 https://github.com/STEllAR-GROUP/hpx/pull/1779
1988 https://github.com/STEllAR-GROUP/hpx/pull/1778
1989 https://github.com/STEllAR-GROUP/hpx/pull/1777
1990 https://github.com/STEllAR-GROUP/hpx/pull/1776
1991 https://github.com/STEllAR-GROUP/hpx/pull/1775
1992 https://github.com/STEllAR-GROUP/hpx/pull/1774
1993 https://github.com/STEllAR-GROUP/hpx/pull/1773
1994 https://github.com/STEllAR-GROUP/hpx/pull/1772
1995 https://github.com/STEllAR-GROUP/hpx/pull/1771
1996 https://github.com/STEllAR-GROUP/hpx/pull/1770
1997 https://github.com/STEllAR-GROUP/hpx/pull/1769
1998 https://github.com/STEllAR-GROUP/hpx/pull/1768
1999 https://github.com/STEllAR-GROUP/hpx/pull/1767
2000 https://github.com/STEllAR-GROUP/hpx/pull/1766
2001 https://github.com/STEllAR-GROUP/hpx/pull/1765
2002 https://github.com/STEllAR-GROUP/hpx/pull/1764
2003 https://github.com/STEllAR-GROUP/hpx/issues/1763
2004 https://github.com/STEllAR-GROUP/hpx/pull/1762
2005 https://github.com/STEllAR-GROUP/hpx/pull/1761
2006 https://github.com/STEllAR-GROUP/hpx/pull/1758

- Issue #1757[2007] - pinning not correct using –hpx:bind
- Issue #1756[2008] - compilation error with MinGW
- PR #1755[2009] - Making output serialization const-correct
- Issue #1753[2010] - HPX performance degrades with time since execution begins
- Issue #1752[2011] - Error in AGAS
- Issue #1751[2012] - hpx::future::wait_for fails a simple test
- PR #1750[2013] - Removing hpx_fwd.hpp includes
- PR #1749[2014] - Simplify result_of and friends
- PR #1747[2015] - Removed superfluous code from message_buffer.hpp
- PR #1746[2016] - Tuple dependencies
- Issue #1745[2017] - Broken when_some which takes iterators
- PR #1744[2018] - Refining archive interface
- PR #1743[2019] - Fixing when_all when only a single future is passed
- PR #1742[2020] - Config includes
- PR #1741[2021] - Os executors
- Issue #1740[2022] - hpx::promise has some problems
- PR #1739[2023] - Parallel composition with generic containers
- Issue #1738[2024] - After building program and successfully linking to a version of hpx DHPX_DIR seems to be ignored
- Issue #1737[2025] - Uptime problems
- PR #1736[2026] - added convenience c-tor and begin()/end() to serialize_buffer
- PR #1735[2027] - Config includes
- PR #1734[2028] - Fixed #1688: Add timer counters for tfunc_total and exec_total
- Issue #1733[2029] - Add unit test for hpx/lcos/local/receive_buffer.hpp

[2007] https://github.com/STEllAR-GROUP/hpx/issues/1757
[2008] https://github.com/STEllAR-GROUP/hpx/issues/1756
[2009] https://github.com/STEllAR-GROUP/hpx/pull/1755
[2010] https://github.com/STEllAR-GROUP/hpx/issues/1753
[2011] https://github.com/STEllAR-GROUP/hpx/issues/1752
[2012] https://github.com/STEllAR-GROUP/hpx/issues/1751
[2013] https://github.com/STEllAR-GROUP/hpx/pull/1750
[2014] https://github.com/STEllAR-GROUP/hpx/pull/1749
[2015] https://github.com/STEllAR-GROUP/hpx/pull/1747
[2016] https://github.com/STEllAR-GROUP/hpx/pull/1746
[2017] https://github.com/STEllAR-GROUP/hpx/issues/1745
[2018] https://github.com/STEllAR-GROUP/hpx/pull/1744
[2019] https://github.com/STEllAR-GROUP/hpx/pull/1743
[2020] https://github.com/STEllAR-GROUP/hpx/pull/1742
[2021] https://github.com/STEllAR-GROUP/hpx/pull/1741
[2022] https://github.com/STEllAR-GROUP/hpx/issues/1740
[2023] https://github.com/STEllAR-GROUP/hpx/pull/1739
[2024] https://github.com/STEllAR-GROUP/hpx/issues/1738
[2025] https://github.com/STEllAR-GROUP/hpx/issues/1737
[2026] https://github.com/STEllAR-GROUP/hpx/pull/1736
[2027] https://github.com/STEllAR-GROUP/hpx/pull/1735
[2028] https://github.com/STEllAR-GROUP/hpx/pull/1734
[2029] https://github.com/STEllAR-GROUP/hpx/issues/1733

- PR #1732[2030] - Renaming get_os_thread_count
- PR #1731[2031] - Basename registration
- Issue #1730[2032] - Use after move of thread_init_data
- PR #1729[2033] - Rewriting channel based on new gate component
- PR #1728[2034] - Fixing #1722
- PR #1727[2035] - Fixing compile problems with apply_colocated
- PR #1726[2036] - Apex integration
- PR #1725[2037] - fixed test timeouts
- PR #1724[2038] - Renaming vector
- Issue #1723[2039] - Drop support for intel compilers and gcc 4.4. based standard libs
- Issue #1722[2040] - Add support for detecting non-ready futures before serialization
- PR #1721[2041] - Unifying parallel executors, initializing from launch policy
- PR #1720[2042] - dropped superfluous typedef
- Issue #1718[2043] - Windows 10 x64, VS 2015 - Unknown CMake command "add_hpx_pseudo_target".
- PR #1717[2044] - Timed executor traits for thread-executors
- PR #1716[2045] - serialization of arrays didn't work with non-pod types. fixed
- PR #1715[2046] - List serialization
- PR #1714[2047] - changing misspellings
- PR #1713[2048] - Fixed distribution policy executors
- PR #1712[2049] - Moving library detection to be executed after feature tests
- PR #1711[2050] - Simplify parcel
- PR #1710[2051] - Compile only tests
- PR #1709[2052] - Implemented timed executors

2030 https://github.com/STEllAR-GROUP/hpx/pull/1732
2031 https://github.com/STEllAR-GROUP/hpx/pull/1731
2032 https://github.com/STEllAR-GROUP/hpx/issues/1730
2033 https://github.com/STEllAR-GROUP/hpx/pull/1729
2034 https://github.com/STEllAR-GROUP/hpx/pull/1728
2035 https://github.com/STEllAR-GROUP/hpx/pull/1727
2036 https://github.com/STEllAR-GROUP/hpx/pull/1726
2037 https://github.com/STEllAR-GROUP/hpx/pull/1725
2038 https://github.com/STEllAR-GROUP/hpx/pull/1724
2039 https://github.com/STEllAR-GROUP/hpx/issues/1723
2040 https://github.com/STEllAR-GROUP/hpx/issues/1722
2041 https://github.com/STEllAR-GROUP/hpx/pull/1721
2042 https://github.com/STEllAR-GROUP/hpx/pull/1720
2043 https://github.com/STEllAR-GROUP/hpx/issues/1718
2044 https://github.com/STEllAR-GROUP/hpx/pull/1717
2045 https://github.com/STEllAR-GROUP/hpx/pull/1716
2046 https://github.com/STEllAR-GROUP/hpx/pull/1715
2047 https://github.com/STEllAR-GROUP/hpx/pull/1714
2048 https://github.com/STEllAR-GROUP/hpx/pull/1713
2049 https://github.com/STEllAR-GROUP/hpx/pull/1712
2050 https://github.com/STEllAR-GROUP/hpx/pull/1711
2051 https://github.com/STEllAR-GROUP/hpx/pull/1710
2052 https://github.com/STEllAR-GROUP/hpx/pull/1709

- PR #1708[2053] - Implement parallel::executor_traits for thread-executors
- PR #1707[2054] - Various fixes to threads::executors to make custom schedulers work
- PR #1706[2055] - Command line option –hpx:cores does not work as expected
- Issue #1705[2056] - command line option –hpx:cores does not work as expected
- PR #1704[2057] - vector deserialization is speeded up a little
- PR #1703[2058] - Fixing shared_mutes
- Issue #1702[2059] - Shared_mutex does not compile with no_mutex cond_var
- PR #1701[2060] - Add distribution_policy_executor
- PR #1700[2061] - Executor parameters
- PR #1699[2062] - Readers writer lock
- PR #1698[2063] - Remove leftovers
- PR #1697[2064] - Fixing held locks
- PR #1696[2065] - Modified Scan Partitioner for Algorithms
- PR #1695[2066] - This thread executors
- PR #1694[2067] - Fixed #1688: Add timer counters for tfunc_total and exec_total
- PR #1693[2068] - Fix #1691: is_executor template specification fails for inherited executors
- PR #1692[2069] - Fixed #1662: Possible exception source in coalescing_message_handler
- Issue #1691[2070] - is_executor template specification fails for inherited executors
- PR #1690[2071] - added macro for non-intrusive serialization of classes without a default c-tor
- PR #1689[2072] - Replace value_or_error with custom storage, unify future_data state
- Issue #1688[2073] - Add timer counters for tfunc_total and exec_total
- PR #1687[2074] - Fixed interval timer
- PR #1686[2075] - Fixing cmake warnings about not existing pseudo target dependencies

[2053] https://github.com/STEllAR-GROUP/hpx/pull/1708
[2054] https://github.com/STEllAR-GROUP/hpx/pull/1707
[2055] https://github.com/STEllAR-GROUP/hpx/pull/1706
[2056] https://github.com/STEllAR-GROUP/hpx/issues/1705
[2057] https://github.com/STEllAR-GROUP/hpx/pull/1704
[2058] https://github.com/STEllAR-GROUP/hpx/pull/1703
[2059] https://github.com/STEllAR-GROUP/hpx/issues/1702
[2060] https://github.com/STEllAR-GROUP/hpx/pull/1701
[2061] https://github.com/STEllAR-GROUP/hpx/pull/1700
[2062] https://github.com/STEllAR-GROUP/hpx/pull/1699
[2063] https://github.com/STEllAR-GROUP/hpx/pull/1698
[2064] https://github.com/STEllAR-GROUP/hpx/pull/1697
[2065] https://github.com/STEllAR-GROUP/hpx/pull/1696
[2066] https://github.com/STEllAR-GROUP/hpx/pull/1695
[2067] https://github.com/STEllAR-GROUP/hpx/pull/1694
[2068] https://github.com/STEllAR-GROUP/hpx/pull/1693
[2069] https://github.com/STEllAR-GROUP/hpx/pull/1692
[2070] https://github.com/STEllAR-GROUP/hpx/issues/1691
[2071] https://github.com/STEllAR-GROUP/hpx/pull/1690
[2072] https://github.com/STEllAR-GROUP/hpx/pull/1689
[2073] https://github.com/STEllAR-GROUP/hpx/issues/1688
[2074] https://github.com/STEllAR-GROUP/hpx/pull/1687
[2075] https://github.com/STEllAR-GROUP/hpx/pull/1686

- PR #1685[2076] - Converting partitioners to use bulk async execute
- PR #1683[2077] - Adds a tool for inspect that checks for character limits
- PR #1682[2078] - Change project name to (uppercase) HPX
- PR #1681[2079] - Counter shortnames
- PR #1680[2080] - Extended Non-intrusive Serialization to Ease Usage for Library Developers
- PR #1679[2081] - Working on 1544: More executor changes
- PR #1678[2082] - Transpose fixes
- PR #1677[2083] - Improve Boost compatibility check
- PR #1676[2084] - 1d stencil fix
- Issue #1675[2085] - hpx project name is not HPX
- PR #1674[2086] - Fixing the MPI parcelport
- PR #1673[2087] - added move semantics to map/vector deserialization
- PR #1672[2088] - Vs2015 await
- PR #1671[2089] - Adapt transform for #1668
- PR #1670[2090] - Started to work on #1668
- PR #1669[2091] - Add this_thread_executors
- Issue #1667[2092] - Apple build instructions in docs are out of date
- PR #1666[2093] - Apex integration
- PR #1665[2094] - Fixes an error with the whitespace check that showed the incorrect location of the error
- Issue #1664[2095] - Inspect tool found incorrect endline whitespace
- PR #1663[2096] - Improve use of locks
- Issue #1662[2097] - Possible exception source in coalescing_message_handler
- PR #1661[2098] - Added support for 128bit number serialization

[2076] https://github.com/STEllAR-GROUP/hpx/pull/1685
[2077] https://github.com/STEllAR-GROUP/hpx/pull/1683
[2078] https://github.com/STEllAR-GROUP/hpx/pull/1682
[2079] https://github.com/STEllAR-GROUP/hpx/pull/1681
[2080] https://github.com/STEllAR-GROUP/hpx/pull/1680
[2081] https://github.com/STEllAR-GROUP/hpx/pull/1679
[2082] https://github.com/STEllAR-GROUP/hpx/pull/1678
[2083] https://github.com/STEllAR-GROUP/hpx/pull/1677
[2084] https://github.com/STEllAR-GROUP/hpx/pull/1676
[2085] https://github.com/STEllAR-GROUP/hpx/issues/1675
[2086] https://github.com/STEllAR-GROUP/hpx/pull/1674
[2087] https://github.com/STEllAR-GROUP/hpx/pull/1673
[2088] https://github.com/STEllAR-GROUP/hpx/pull/1672
[2089] https://github.com/STEllAR-GROUP/hpx/pull/1671
[2090] https://github.com/STEllAR-GROUP/hpx/pull/1670
[2091] https://github.com/STEllAR-GROUP/hpx/pull/1669
[2092] https://github.com/STEllAR-GROUP/hpx/issues/1667
[2093] https://github.com/STEllAR-GROUP/hpx/pull/1666
[2094] https://github.com/STEllAR-GROUP/hpx/pull/1665
[2095] https://github.com/STEllAR-GROUP/hpx/issues/1664
[2096] https://github.com/STEllAR-GROUP/hpx/pull/1663
[2097] https://github.com/STEllAR-GROUP/hpx/issues/1662
[2098] https://github.com/STEllAR-GROUP/hpx/pull/1661

- PR #1660[2099] - Serialization 128bits
- PR #1659[2100] - Implemented inner_product and adjacent_diff algos
- PR #1658[2101] - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1657[2102] - Use of shared_ptr in io_service_pool changed to unique_ptr
- Issue #1656[2103] - 1d_stencil codes all have wrong factor
- PR #1654[2104] - When using runtime_mode_connect, find the correct localhost public ip address
- PR #1653[2105] - Fixing 1617
- PR #1652[2106] - Remove traits::action_may_require_id_splitting
- PR #1651[2107] - Fixed performance counters related to AGAS cache timings
- PR #1650[2108] - Remove leftovers of traits::type_size
- PR #1649[2109] - Shorten target names on Windows to shorten used path names
- PR #1648[2110] - Fixing problems introduced by merging #1623 for older compilers
- PR #1647[2111] - Simplify running automatic builds on Windows
- Issue #1646[2112] - Cache insert and update performance counters are broken
- Issue #1644[2113] - Remove leftovers of traits::type_size
- Issue #1643[2114] - Remove traits::action_may_require_id_splitting
- PR #1642[2115] - Adds spell checker to the inspect tool for qbk and doxygen comments
- PR #1640[2116] - First step towards fixing 688
- PR #1639[2117] - Re-apply remaining changes from limit_dataflow_recursion branch
- PR #1638[2118] - This fixes possible deadlock in the test ignore_while_locked_1485
- PR #1637[2119] - Fixing hpx::wait_all() invoked with two vector<future<T>>
- PR #1636[2120] - Partially re-apply changes from limit_dataflow_recursion branch
- PR #1635[2121] - Adding missing test for #1572

[2099] https://github.com/STEllAR-GROUP/hpx/pull/1660
[2100] https://github.com/STEllAR-GROUP/hpx/pull/1659
[2101] https://github.com/STEllAR-GROUP/hpx/pull/1658
[2102] https://github.com/STEllAR-GROUP/hpx/pull/1657
[2103] https://github.com/STEllAR-GROUP/hpx/issues/1656
[2104] https://github.com/STEllAR-GROUP/hpx/pull/1654
[2105] https://github.com/STEllAR-GROUP/hpx/pull/1653
[2106] https://github.com/STEllAR-GROUP/hpx/pull/1652
[2107] https://github.com/STEllAR-GROUP/hpx/pull/1651
[2108] https://github.com/STEllAR-GROUP/hpx/pull/1650
[2109] https://github.com/STEllAR-GROUP/hpx/pull/1649
[2110] https://github.com/STEllAR-GROUP/hpx/pull/1648
[2111] https://github.com/STEllAR-GROUP/hpx/pull/1647
[2112] https://github.com/STEllAR-GROUP/hpx/issues/1646
[2113] https://github.com/STEllAR-GROUP/hpx/issues/1644
[2114] https://github.com/STEllAR-GROUP/hpx/issues/1643
[2115] https://github.com/STEllAR-GROUP/hpx/pull/1642
[2116] https://github.com/STEllAR-GROUP/hpx/pull/1640
[2117] https://github.com/STEllAR-GROUP/hpx/pull/1639
[2118] https://github.com/STEllAR-GROUP/hpx/pull/1638
[2119] https://github.com/STEllAR-GROUP/hpx/pull/1637
[2120] https://github.com/STEllAR-GROUP/hpx/pull/1636
[2121] https://github.com/STEllAR-GROUP/hpx/pull/1635

- PR #1634[2122] - Revert "Limit recursion-depth in dataflow to a configurable constant"
- PR #1633[2123] - Add command line option to ignore batch environment
- PR #1631[2124] - hpx::lcos::queue exhibits strange behavior
- PR #1630[2125] - Fixed endline_whitespace_check.cpp to detect lines with only whitespace
- Issue #1629[2126] - Inspect trailing whitespace checker problem
- PR #1628[2127] - Removed meaningless const qualifiers. Minor icpc fix.
- PR #1627[2128] - Fixing the queue LCO and add example demonstrating its use
- PR #1626[2129] - Deprecating get_gid(), add get_id() and get_unmanaged_id()
- PR #1625[2130] - Allowing to specify whether to send credits along with message
- Issue #1624[2131] - Lifetime issue
- Issue #1623[2132] - hpx::wait_all() invoked with two vector<future<T>> fails
- PR #1622[2133] - Executor partitioners
- PR #1621[2134] - Clean up coroutines implementation
- Issue #1620[2135] - Revert #1535
- PR #1619[2136] - Fix result type calculation for hpx::make_continuation
- PR #1618[2137] - Fixing RDTSC on Xeon/Phi
- Issue #1617[2138] - hpx cmake not working when run as a subproject
- Issue #1616[2139] - cmake problem resulting in RDTSC not working correctly for Xeon Phi creates very strange results for duration counters
- Issue #1615[2140] - hpx::make_continuation requires input and output to be the same
- PR #1614[2141] - Fixed remove copy test
- Issue #1613[2142] - Dataflow causes stack overflow
- PR #1612[2143] - Modified foreach partitioner to use bulk execute
- PR #1611[2144] - Limit recursion-depth in dataflow to a configurable constant

2122 https://github.com/STEllAR-GROUP/hpx/pull/1634
2123 https://github.com/STEllAR-GROUP/hpx/pull/1633
2124 https://github.com/STEllAR-GROUP/hpx/pull/1631
2125 https://github.com/STEllAR-GROUP/hpx/pull/1630
2126 https://github.com/STEllAR-GROUP/hpx/issues/1629
2127 https://github.com/STEllAR-GROUP/hpx/pull/1628
2128 https://github.com/STEllAR-GROUP/hpx/pull/1627
2129 https://github.com/STEllAR-GROUP/hpx/pull/1626
2130 https://github.com/STEllAR-GROUP/hpx/pull/1625
2131 https://github.com/STEllAR-GROUP/hpx/issues/1624
2132 https://github.com/STEllAR-GROUP/hpx/issues/1623
2133 https://github.com/STEllAR-GROUP/hpx/pull/1622
2134 https://github.com/STEllAR-GROUP/hpx/pull/1621
2135 https://github.com/STEllAR-GROUP/hpx/issues/1620
2136 https://github.com/STEllAR-GROUP/hpx/pull/1619
2137 https://github.com/STEllAR-GROUP/hpx/pull/1618
2138 https://github.com/STEllAR-GROUP/hpx/issues/1617
2139 https://github.com/STEllAR-GROUP/hpx/issues/1616
2140 https://github.com/STEllAR-GROUP/hpx/issues/1615
2141 https://github.com/STEllAR-GROUP/hpx/pull/1614
2142 https://github.com/STEllAR-GROUP/hpx/issues/1613
2143 https://github.com/STEllAR-GROUP/hpx/pull/1612
2144 https://github.com/STEllAR-GROUP/hpx/pull/1611

- PR #1610[2145] - Increase timeout for CircleCI
- PR #1609[2146] - Refactoring thread manager, mainly extracting thread pool
- PR #1608[2147] - Fixed running multiple localities without localities parameter
- PR #1607[2148] - More algorithm fixes to adjacentfind
- Issue #1606[2149] - Running without localities parameter binds to bogus port range
- Issue #1605[2150] - Too many serializations
- PR #1604[2151] - Changes the HPX image into a hyperlink
- PR #1601[2152] - Fixing problems with remove_copy algorithm tests
- PR #1600[2153] - Actions with ids cleanup
- PR #1599[2154] - Duplicate binding of global ids should fail
- PR #1598[2155] - Fixing array access
- PR #1597[2156] - Improved the reliability of connecting/disconnecting localities
- Issue #1596[2157] - Duplicate id binding should fail
- PR #1595[2158] - Fixing more cmake config constants
- PR #1594[2159] - Fixing preprocessor constant used to enable C++11 chrono
- PR #1593[2160] - Adding operator|() for hpx::launch
- Issue #1592[2161] - Error (typo) in the docs
- Issue #1590[2162] - CMake fails when CMAKE_BINARY_DIR contains '+'.
- Issue #1589[2163] - Disconnecting a locality results in segfault using heartbeat example
- PR #1588[2164] - Fix doc string for config option HPX_WITH_EXAMPLES
- PR #1586[2165] - Fixing 1493
- PR #1585[2166] - Additional Check for Inspect Tool to detect Endline Whitespace
- Issue #1584[2167] - Clean up coroutines implementation

2145 https://github.com/STEllAR-GROUP/hpx/pull/1610
2146 https://github.com/STEllAR-GROUP/hpx/pull/1609
2147 https://github.com/STEllAR-GROUP/hpx/pull/1608
2148 https://github.com/STEllAR-GROUP/hpx/pull/1607
2149 https://github.com/STEllAR-GROUP/hpx/issues/1606
2150 https://github.com/STEllAR-GROUP/hpx/issues/1605
2151 https://github.com/STEllAR-GROUP/hpx/pull/1604
2152 https://github.com/STEllAR-GROUP/hpx/pull/1601
2153 https://github.com/STEllAR-GROUP/hpx/pull/1600
2154 https://github.com/STEllAR-GROUP/hpx/pull/1599
2155 https://github.com/STEllAR-GROUP/hpx/pull/1598
2156 https://github.com/STEllAR-GROUP/hpx/pull/1597
2157 https://github.com/STEllAR-GROUP/hpx/issues/1596
2158 https://github.com/STEllAR-GROUP/hpx/pull/1595
2159 https://github.com/STEllAR-GROUP/hpx/pull/1594
2160 https://github.com/STEllAR-GROUP/hpx/pull/1593
2161 https://github.com/STEllAR-GROUP/hpx/issues/1592
2162 https://github.com/STEllAR-GROUP/hpx/issues/1590
2163 https://github.com/STEllAR-GROUP/hpx/issues/1589
2164 https://github.com/STEllAR-GROUP/hpx/pull/1588
2165 https://github.com/STEllAR-GROUP/hpx/pull/1586
2166 https://github.com/STEllAR-GROUP/hpx/pull/1585
2167 https://github.com/STEllAR-GROUP/hpx/issues/1584

- PR #1583[2168] - Adding a check for end line whitespace
- PR #1582[2169] - Attempt to fix assert firing after scheduling loop was exited
- PR #1581[2170] - Fixed adjacentfind_binary test
- PR #1580[2171] - Prevent some of the internal cmake lists from growing indefinitely
- PR #1579[2172] - Removing type_size trait, replacing it with special archive type
- Issue #1578[2173] - Remove demangle_helper
- PR #1577[2174] - Get ptr problems
- Issue #1576[2175] - Refactor async, dataflow, and future::then
- PR #1575[2176] - Fixing tests for parallel rotate
- PR #1574[2177] - Cleaning up schedulers
- PR #1573[2178] - Fixing thread pool executor
- PR #1572[2179] - Fixing number of configured localities
- PR #1571[2180] - Reimplement decay
- PR #1570[2181] - Refactoring async, apply, and dataflow APIs
- PR #1569[2182] - Changed range for mach-o library lookup
- PR #1568[2183] - Mark decltype support as required
- PR #1567[2184] - Removed const from algorithms
- Issue #1566[2185] - CMAKE Configuration Test Failures for clang 3.5 on debian
- PR #1565[2186] - Dylib support
- PR #1564[2187] - Converted partitioners and some algorithms to use executors
- PR #1563[2188] - Fix several #includes for Boost.Preprocessor
- PR #1562[2189] - Adding configuration option disabling/enabling all message handlers
- PR #1561[2190] - Removed all occurrences of boost::move replacing it with std::move

[2168] https://github.com/STEllAR-GROUP/hpx/pull/1583
[2169] https://github.com/STEllAR-GROUP/hpx/pull/1582
[2170] https://github.com/STEllAR-GROUP/hpx/pull/1581
[2171] https://github.com/STEllAR-GROUP/hpx/pull/1580
[2172] https://github.com/STEllAR-GROUP/hpx/pull/1579
[2173] https://github.com/STEllAR-GROUP/hpx/issues/1578
[2174] https://github.com/STEllAR-GROUP/hpx/pull/1577
[2175] https://github.com/STEllAR-GROUP/hpx/issues/1576
[2176] https://github.com/STEllAR-GROUP/hpx/pull/1575
[2177] https://github.com/STEllAR-GROUP/hpx/pull/1574
[2178] https://github.com/STEllAR-GROUP/hpx/pull/1573
[2179] https://github.com/STEllAR-GROUP/hpx/pull/1572
[2180] https://github.com/STEllAR-GROUP/hpx/pull/1571
[2181] https://github.com/STEllAR-GROUP/hpx/pull/1570
[2182] https://github.com/STEllAR-GROUP/hpx/pull/1569
[2183] https://github.com/STEllAR-GROUP/hpx/pull/1568
[2184] https://github.com/STEllAR-GROUP/hpx/pull/1567
[2185] https://github.com/STEllAR-GROUP/hpx/issues/1566
[2186] https://github.com/STEllAR-GROUP/hpx/pull/1565
[2187] https://github.com/STEllAR-GROUP/hpx/pull/1564
[2188] https://github.com/STEllAR-GROUP/hpx/pull/1563
[2189] https://github.com/STEllAR-GROUP/hpx/pull/1562
[2190] https://github.com/STEllAR-GROUP/hpx/pull/1561

- Issue #1560[2191] - Leftover HPX_REGISTER_ACTION_DECLARATION_2
- PR #1558[2192] - Revisit async/apply SFINAE conditions
- PR #1557[2193] - Removing type_size trait, replacing it with special archive type
- PR #1556[2194] - Executor algorithms
- PR #1555[2195] - Remove the necessity to specify archive flags on the receiving end
- PR #1554[2196] - Removing obsolete Boost.Serialization macros
- PR #1553[2197] - Properly fix HPX_DEFINE_*_ACTION macros
- PR #1552[2198] - Fixed algorithms relying on copy_if implementation
- PR #1551[2199] - Pxfs - Modifying FindOrangeFS.cmake based on OrangeFS 2.9.X
- Issue #1550[2200] - Passing plain identifier inside HPX_DEFINE_PLAIN_ACTION_1
- PR #1549[2201] - Fixing intel14/libstdc++4.4
- PR #1548[2202] - Moving raw_ptr to detail namespace
- PR #1547[2203] - Adding support for executors to future.then
- PR #1546[2204] - Executor traits result types
- PR #1545[2205] - Integrate executors with dataflow
- PR #1543[2206] - Fix potential zero-copy for primarynamespace::bulk_service_async et.al.
- PR #1542[2207] - Merging HPX0.9.10 into pxfs branch
- PR #1541[2208] - Removed stale cmake tests, unused since the great cmake refactoring
- PR #1540[2209] - Fix idle-rate on platforms without TSC
- PR #1539[2210] - Reporting situation if zero-copy-serialization was performed by a parcel generated from a plain apply/async
- PR #1538[2211] - Changed return type of bulk executors and added test
- Issue #1537[2212] - Incorrect cpuid config tests
- PR #1536[2213] - Changed return type of bulk executors and added test

[2191] https://github.com/STEllAR-GROUP/hpx/issues/1560
[2192] https://github.com/STEllAR-GROUP/hpx/pull/1558
[2193] https://github.com/STEllAR-GROUP/hpx/pull/1557
[2194] https://github.com/STEllAR-GROUP/hpx/pull/1556
[2195] https://github.com/STEllAR-GROUP/hpx/pull/1555
[2196] https://github.com/STEllAR-GROUP/hpx/pull/1554
[2197] https://github.com/STEllAR-GROUP/hpx/pull/1553
[2198] https://github.com/STEllAR-GROUP/hpx/pull/1552
[2199] https://github.com/STEllAR-GROUP/hpx/pull/1551
[2200] https://github.com/STEllAR-GROUP/hpx/issues/1550
[2201] https://github.com/STEllAR-GROUP/hpx/pull/1549
[2202] https://github.com/STEllAR-GROUP/hpx/pull/1548
[2203] https://github.com/STEllAR-GROUP/hpx/pull/1547
[2204] https://github.com/STEllAR-GROUP/hpx/pull/1546
[2205] https://github.com/STEllAR-GROUP/hpx/pull/1545
[2206] https://github.com/STEllAR-GROUP/hpx/pull/1543
[2207] https://github.com/STEllAR-GROUP/hpx/pull/1542
[2208] https://github.com/STEllAR-GROUP/hpx/pull/1541
[2209] https://github.com/STEllAR-GROUP/hpx/pull/1540
[2210] https://github.com/STEllAR-GROUP/hpx/pull/1539
[2211] https://github.com/STEllAR-GROUP/hpx/pull/1538
[2212] https://github.com/STEllAR-GROUP/hpx/issues/1537
[2213] https://github.com/STEllAR-GROUP/hpx/pull/1536

- PR #1535[2214] - Make sure promise::get_gid() can be called more than once
- PR #1534[2215] - Fixed async_callback with bound callback
- PR #1533[2216] - Updated the link in the documentation to a publically- accessible URL
- PR #1532[2217] - Make sure sync primitives are not copyable nor movable
- PR #1531[2218] - Fix unwrapped issue with future ranges of void type
- PR #1530[2219] - Serialization complex
- Issue #1528[2220] - Unwrapped issue with future<void>
- Issue #1527[2221] - HPX does not build with Boost 1.58.0
- PR #1526[2222] - Added support for boost.multi_array serialization
- PR #1525[2223] - Properly handle deferred futures, fixes #1506
- PR #1524[2224] - Making sure invalid action argument types generate clear error message
- Issue #1522[2225] - Need serialization support for boost multi array
- Issue #1521[2226] - Remote async and zero-copy serialization optimizations don't play well together
- PR #1520[2227] - Fixing UB whil registering polymorphic classes for serialization
- PR #1519[2228] - Making detail::condition_variable safe to use
- PR #1518[2229] - Fix when_some bug missing indices in its result
- Issue #1517[2230] - Typo may affect CMake build system tests
- PR #1516[2231] - Fixing Posix context
- PR #1515[2232] - Fixing Posix context
- PR #1514[2233] - Correct problems with loading dynamic components
- PR #1513[2234] - Fixing intel glibc4 4
- Issue #1508[2235] - memory and papi counters do not work
- Issue #1507[2236] - Unrecognized Command Line Option Error causing exit status 0

2214 https://github.com/STEllAR-GROUP/hpx/pull/1535
2215 https://github.com/STEllAR-GROUP/hpx/pull/1534
2216 https://github.com/STEllAR-GROUP/hpx/pull/1533
2217 https://github.com/STEllAR-GROUP/hpx/pull/1532
2218 https://github.com/STEllAR-GROUP/hpx/pull/1531
2219 https://github.com/STEllAR-GROUP/hpx/pull/1530
2220 https://github.com/STEllAR-GROUP/hpx/issues/1528
2221 https://github.com/STEllAR-GROUP/hpx/issues/1527
2222 https://github.com/STEllAR-GROUP/hpx/pull/1526
2223 https://github.com/STEllAR-GROUP/hpx/pull/1525
2224 https://github.com/STEllAR-GROUP/hpx/pull/1524
2225 https://github.com/STEllAR-GROUP/hpx/issues/1522
2226 https://github.com/STEllAR-GROUP/hpx/issues/1521
2227 https://github.com/STEllAR-GROUP/hpx/pull/1520
2228 https://github.com/STEllAR-GROUP/hpx/pull/1519
2229 https://github.com/STEllAR-GROUP/hpx/pull/1518
2230 https://github.com/STEllAR-GROUP/hpx/issues/1517
2231 https://github.com/STEllAR-GROUP/hpx/pull/1516
2232 https://github.com/STEllAR-GROUP/hpx/pull/1515
2233 https://github.com/STEllAR-GROUP/hpx/pull/1514
2234 https://github.com/STEllAR-GROUP/hpx/pull/1513
2235 https://github.com/STEllAR-GROUP/hpx/issues/1508
2236 https://github.com/STEllAR-GROUP/hpx/issues/1507

- Issue #1506[2237] - Properly handle deferred futures
- PR #1505[2238] - Adding #include - would not compile without this
- Issue #1502[2239] - `boost::filesystem::exists` throws unexpected exception
- Issue #1501[2240] - hwloc configuration options are wrong for MIC
- PR #1504[2241] - Making sure boost::filesystem::exists() does not throw
- PR #1500[2242] - Exit application on `--hpx:version/-v` and `--hpx:info`
- PR #1498[2243] - Extended task block
- PR #1497[2244] - Unique ptr serialization
- PR #1496[2245] - Unique ptr serialization (closed)
- PR #1495[2246] - Switching circleci build type to debug
- Issue #1494[2247] - `--hpx:version/-v` does not exit after printing version information
- Issue #1493[2248] - add an `hpx_` prefix to libraries and components to avoid name conflicts
- Issue #1492[2249] - Define and ensure limitations for arguments to async/apply
- PR #1489[2250] - Enable idle rate counter on demand
- PR #1488[2251] - Made sure `detail::condition_variable` can be safely destroyed
- PR #1487[2252] - Introduced default (main) template implementation for `ignore_while_checking`
- PR #1486[2253] - Add HPX inspect tool
- Issue #1485[2254] - `ignore_while_locked` doesn't support all Lockable types
- PR #1484[2255] - Docker image generation
- PR #1483[2256] - Move external endian library into HPX
- PR #1482[2257] - Actions with integer type ids
- Issue #1481[2258] - Sync primitives safe destruction
- Issue #1480[2259] - Move external/boost/endian into hpx/util

[2237] https://github.com/STEllAR-GROUP/hpx/issues/1506
[2238] https://github.com/STEllAR-GROUP/hpx/pull/1505
[2239] https://github.com/STEllAR-GROUP/hpx/issues/1502
[2240] https://github.com/STEllAR-GROUP/hpx/issues/1501
[2241] https://github.com/STEllAR-GROUP/hpx/pull/1504
[2242] https://github.com/STEllAR-GROUP/hpx/pull/1500
[2243] https://github.com/STEllAR-GROUP/hpx/pull/1498
[2244] https://github.com/STEllAR-GROUP/hpx/pull/1497
[2245] https://github.com/STEllAR-GROUP/hpx/pull/1496
[2246] https://github.com/STEllAR-GROUP/hpx/pull/1495
[2247] https://github.com/STEllAR-GROUP/hpx/issues/1494
[2248] https://github.com/STEllAR-GROUP/hpx/issues/1493
[2249] https://github.com/STEllAR-GROUP/hpx/issues/1492
[2250] https://github.com/STEllAR-GROUP/hpx/pull/1489
[2251] https://github.com/STEllAR-GROUP/hpx/pull/1488
[2252] https://github.com/STEllAR-GROUP/hpx/pull/1487
[2253] https://github.com/STEllAR-GROUP/hpx/pull/1486
[2254] https://github.com/STEllAR-GROUP/hpx/issues/1485
[2255] https://github.com/STEllAR-GROUP/hpx/pull/1484
[2256] https://github.com/STEllAR-GROUP/hpx/pull/1483
[2257] https://github.com/STEllAR-GROUP/hpx/pull/1482
[2258] https://github.com/STEllAR-GROUP/hpx/issues/1481
[2259] https://github.com/STEllAR-GROUP/hpx/issues/1480

- Issue #1478[2260] - Boost inspect violations
- PR #1479[2261] - Adds serialization for arrays; some futher/minor fixes
- PR #1477[2262] - Fixing problems with the Intel compiler using a GCC 4.4 std library
- PR #1476[2263] - Adding `hpx::lcos::latch` and `hpx::lcos::local::latch`
- Issue #1475[2264] - Boost inspect violations
- PR #1473[2265] - Fixing action move tests
- Issue #1471[2266] - Sync primitives should not be movable
- PR #1470[2267] - Removing `hpx::util::polymorphic_factory`
- PR #1468[2268] - Fixed container creation
- Issue #1467[2269] - HPX application fail during finalization
- Issue #1466[2270] - HPX doesn't pick up Torque's nodefile on SuperMIC
- Issue #1464[2271] - HPX option for pre and post bootstrap performance counters
- PR #1463[2272] - Replacing `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1462[2273] - Consolidated task_region with N4411
- PR #1461[2274] - Consolidate inconsistent CMake option names
- Issue #1460[2275] - Which malloc is actually used? or at least which one is HPX built with
- Issue #1459[2276] - Make cmake configure step fail explicitly if compiler version is not supported
- Issue #1458[2277] - Update `parallel::task_region` with N4411
- PR #1456[2278] - Consolidating `new_<>()`
- Issue #1455[2279] - Replace `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1454[2280] - Removed harmful std::moves from return statements
- PR #1453[2281] - Use range-based for-loop instead of Boost.Foreach
- PR #1452[2282] - C++ feature tests

[2260] https://github.com/STEllAR-GROUP/hpx/issues/1478
[2261] https://github.com/STEllAR-GROUP/hpx/pull/1479
[2262] https://github.com/STEllAR-GROUP/hpx/pull/1477
[2263] https://github.com/STEllAR-GROUP/hpx/pull/1476
[2264] https://github.com/STEllAR-GROUP/hpx/issues/1475
[2265] https://github.com/STEllAR-GROUP/hpx/pull/1473
[2266] https://github.com/STEllAR-GROUP/hpx/issues/1471
[2267] https://github.com/STEllAR-GROUP/hpx/pull/1470
[2268] https://github.com/STEllAR-GROUP/hpx/pull/1468
[2269] https://github.com/STEllAR-GROUP/hpx/issues/1467
[2270] https://github.com/STEllAR-GROUP/hpx/issues/1466
[2271] https://github.com/STEllAR-GROUP/hpx/issues/1464
[2272] https://github.com/STEllAR-GROUP/hpx/pull/1463
[2273] https://github.com/STEllAR-GROUP/hpx/pull/1462
[2274] https://github.com/STEllAR-GROUP/hpx/pull/1461
[2275] https://github.com/STEllAR-GROUP/hpx/issues/1460
[2276] https://github.com/STEllAR-GROUP/hpx/issues/1459
[2277] https://github.com/STEllAR-GROUP/hpx/issues/1458
[2278] https://github.com/STEllAR-GROUP/hpx/pull/1456
[2279] https://github.com/STEllAR-GROUP/hpx/issues/1455
[2280] https://github.com/STEllAR-GROUP/hpx/pull/1454
[2281] https://github.com/STEllAR-GROUP/hpx/pull/1453
[2282] https://github.com/STEllAR-GROUP/hpx/pull/1452

- PR #1451[2283] - When serializing, pass archive flags to traits::get_type_size
- Issue #1450[2284] - traits:get_type_size needs archive flags to enable zero_copy optimizations
- Issue #1449[2285] - "couldn't create performance counter" - AGAS
- Issue #1448[2286] - Replace distributing factories with `new_<T[]>(...)`
- PR #1447[2287] - Removing obsolete remote_object component
- PR #1446[2288] - Hpx serialization
- PR #1445[2289] - Replacing travis with circleci
- PR #1443[2290] - Always stripping HPX command line arguments before executing start function
- PR #1442[2291] - Adding –hpx:bind=none to disable thread affinities
- Issue #1439[2292] - Libraries get linked in multiple times, RPATH is not properly set
- PR #1438[2293] - Removed superfluous typedefs
- Issue #1437[2294] - `hpx::init()` should strip HPX-related flags from argv
- Issue #1436[2295] - Add strong scaling option to htts
- PR #1435[2296] - Adding `async_cb`, `async_continue_cb`, and `async_colocated_cb`
- PR #1434[2297] - Added missing install rule, removed some dead CMake code
- PR #1433[2298] - Add GitExternal and SubProject cmake scripts from eyescale/cmake repo
- Issue #1432[2299] - Add command line flag to disable thread pinning
- PR #1431[2300] - Fix #1423
- Issue #1430[2301] - Inconsistent CMake option names
- Issue #1429[2302] - Configure setting `HPX_HAVE_PARCELPORT_MPI` is ignored
- PR #1428[2303] - Fixes #1419 (closed)
- PR #1427[2304] - Adding stencil_iterator and transform_iterator
- PR #1426[2305] - Fixes #1419

2283 https://github.com/STEllAR-GROUP/hpx/pull/1451
2284 https://github.com/STEllAR-GROUP/hpx/issues/1450
2285 https://github.com/STEllAR-GROUP/hpx/issues/1449
2286 https://github.com/STEllAR-GROUP/hpx/issues/1448
2287 https://github.com/STEllAR-GROUP/hpx/pull/1447
2288 https://github.com/STEllAR-GROUP/hpx/pull/1446
2289 https://github.com/STEllAR-GROUP/hpx/pull/1445
2290 https://github.com/STEllAR-GROUP/hpx/pull/1443
2291 https://github.com/STEllAR-GROUP/hpx/pull/1442
2292 https://github.com/STEllAR-GROUP/hpx/issues/1439
2293 https://github.com/STEllAR-GROUP/hpx/pull/1438
2294 https://github.com/STEllAR-GROUP/hpx/issues/1437
2295 https://github.com/STEllAR-GROUP/hpx/issues/1436
2296 https://github.com/STEllAR-GROUP/hpx/pull/1435
2297 https://github.com/STEllAR-GROUP/hpx/pull/1434
2298 https://github.com/STEllAR-GROUP/hpx/pull/1433
2299 https://github.com/STEllAR-GROUP/hpx/issues/1432
2300 https://github.com/STEllAR-GROUP/hpx/pull/1431
2301 https://github.com/STEllAR-GROUP/hpx/issues/1430
2302 https://github.com/STEllAR-GROUP/hpx/issues/1429
2303 https://github.com/STEllAR-GROUP/hpx/pull/1428
2304 https://github.com/STEllAR-GROUP/hpx/pull/1427
2305 https://github.com/STEllAR-GROUP/hpx/pull/1426

- PR #1425[2306] - During serialization memory allocation should honour allocator chunk size
- Issue #1424[2307] - chunk allocation during serialization does not use memory pool/allocator chunk size
- Issue #1423[2308] - Remove `HPX_STD_UNIQUE_PTR`
- Issue #1422[2309] - hpx:threads=all allocates too many os threads
- PR #1420[2310] - added .travis.yml
- Issue #1419[2311] - Unify enums: `hpx::runtime::state` and `hpx::state`
- PR #1416[2312] - Adding travis builder
- Issue #1414[2313] - Correct directory for dispatch_gcc46.hpp iteration
- Issue #1410[2314] - Set operation algorithms
- Issue #1389[2315] - Parallel algorithms relying on scan partitioner break for small number of elements
- Issue #1325[2316] - Exceptions thrown during parcel handling are not handled correctly
- Issue #1315[2317] - Errors while running performance tests
- Issue #1309[2318] - `hpx::vector` partitions are not easily extendable by applications
- PR #1300[2319] - Added serialization/de-serialization to examples.tuplespace
- Issue #1251[2320] - hpx::threads::get_thread_count doesn't consider pending threads
- Issue #1008[2321] - Decrease in application performance overtime; occasional spikes of major slowdown
- Issue #1001[2322] - Zero copy serialization raises assert
- Issue #721[2323] - Make HPX usable for Xeon Phi
- Issue #524[2324] - Extend scheduler to support threads which can't be stolen

2.10.8 *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.

2306 https://github.com/STEllAR-GROUP/hpx/pull/1425
2307 https://github.com/STEllAR-GROUP/hpx/issues/1424
2308 https://github.com/STEllAR-GROUP/hpx/issues/1423
2309 https://github.com/STEllAR-GROUP/hpx/issues/1422
2310 https://github.com/STEllAR-GROUP/hpx/pull/1420
2311 https://github.com/STEllAR-GROUP/hpx/issues/1419
2312 https://github.com/STEllAR-GROUP/hpx/pull/1416
2313 https://github.com/STEllAR-GROUP/hpx/issues/1414
2314 https://github.com/STEllAR-GROUP/hpx/issues/1410
2315 https://github.com/STEllAR-GROUP/hpx/issues/1389
2316 https://github.com/STEllAR-GROUP/hpx/issues/1325
2317 https://github.com/STEllAR-GROUP/hpx/issues/1315
2318 https://github.com/STEllAR-GROUP/hpx/issues/1309
2319 https://github.com/STEllAR-GROUP/hpx/pull/1300
2320 https://github.com/STEllAR-GROUP/hpx/issues/1251
2321 https://github.com/STEllAR-GROUP/hpx/issues/1008
2322 https://github.com/STEllAR-GROUP/hpx/issues/1001
2323 https://github.com/STEllAR-GROUP/hpx/issues/721
2324 https://github.com/STEllAR-GROUP/hpx/issues/524

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[2325]).

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[2326] for a documentation of our progress).

Breaking changes

With this release we put a lot of effort into changing the code base to be more compatible to C++11. These changes have caused the following issues for backward compatibility:

- Move to Variadics- All of the API now uses variadic templates. However, this change required to modify the argument sequence for some of the exiting API functions (`hpx::async_continue`, `hpx::apply_continue`, *`hpx::when_each`*, *`hpx::wait_each`*, synchronous invocation of actions).
- Changes to Macros- We also removed the macros `HPX_STD_FUNCTION` and `HPX_STD_TUPLE`. This shouldn't affect any user code as we replaced `HPX_STD_FUNCTION` with `hpx::util::function_nonser` which was the default expansion used for this macro. All *HPX* API functions which expect a `hpx::util::function_nonser` (or a `hpx::util::unique_function_nonser`) can now be transparently called with a compatible `std::function` instead. Similarly, `HPX_STD_TUPLE` was replaced by its default expansion as well: `hpx::util::tuple`.
- Changes to `hpx::unique_future`- `hpx::unique_future`, which was deprecated in the previous release for `hpx::future` is now completely removed from *HPX*. This completes the transition to a completely standards conforming implementation of `hpx::future`.
- Changes to Supported Compilers. Finally, in order to utilize more C++11 semantics, we have officially dropped support for GCC 4.4 and MSVC 2012. Please see our *Prerequisites* page for more details.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

2325 https://github.com/STEllAR-GROUP/hpx/issues/839
2326 https://github.com/STEllAR-GROUP/hpx/issues/1338

- Issue #1402[2327] - Internal shared_future serialization copies
- Issue #1399[2328] - Build takes unusually long time…
- Issue #1398[2329] - Tests using the scan partitioner are broken on at least gcc 4.7 and intel compiler
- Issue #1397[2330] - Completely remove hpx::unique_future
- Issue #1396[2331] - Parallel scan algorithms with different initial values
- Issue #1395[2332] - Race Condition - 1d_stencil_8 - SuperMIC
- Issue #1394[2333] - "suspending thread while at least one lock is being held" - 1d_stencil_8 - SuperMIC
- Issue #1393[2334] - SEGFAULT in 1d_stencil_8 on SuperMIC
- Issue #1392[2335] - Fixing #1168
- Issue #1391[2336] - Parallel Algorithms for scan partitioner for small number of elements
- Issue #1387[2337] - Failure with more than 4 localities
- Issue #1386[2338] - Dispatching unhandled exceptions to outer user code
- Issue #1385[2339] - Adding Copy algorithms, fixing `parallel::copy_if`
- Issue #1384[2340] - Fixing 1325
- Issue #1383[2341] - Fixed #504: Refactor Dataflow LCO to work with futures, this removes the dataflow component as it is obsolete
- Issue #1382[2342] - `is_sorted`, `is_sorted_until` and `is_partitioned` algorithms
- Issue #1381[2343] - fix for CMake versions prior to 3.1
- Issue #1380[2344] - resolved warning in CMake 3.1 and newer
- Issue #1379[2345] - Compilation error with papi
- Issue #1378[2346] - Towards safer migration
- Issue #1377[2347] - HPXConfig.cmake should include `TCMALLOC_LIBRARY` and `TCMALLOC_INCLUDE_DIR`
- Issue #1376[2348] - Warning on uninitialized member
- Issue #1375[2349] - Fixing 1163

[2327] https://github.com/STEllAR-GROUP/hpx/issues/1402
[2328] https://github.com/STEllAR-GROUP/hpx/issues/1399
[2329] https://github.com/STEllAR-GROUP/hpx/issues/1398
[2330] https://github.com/STEllAR-GROUP/hpx/issues/1397
[2331] https://github.com/STEllAR-GROUP/hpx/issues/1396
[2332] https://github.com/STEllAR-GROUP/hpx/issues/1395
[2333] https://github.com/STEllAR-GROUP/hpx/issues/1394
[2334] https://github.com/STEllAR-GROUP/hpx/issues/1393
[2335] https://github.com/STEllAR-GROUP/hpx/issues/1392
[2336] https://github.com/STEllAR-GROUP/hpx/issues/1391
[2337] https://github.com/STEllAR-GROUP/hpx/issues/1387
[2338] https://github.com/STEllAR-GROUP/hpx/issues/1386
[2339] https://github.com/STEllAR-GROUP/hpx/issues/1385
[2340] https://github.com/STEllAR-GROUP/hpx/issues/1384
[2341] https://github.com/STEllAR-GROUP/hpx/issues/1383
[2342] https://github.com/STEllAR-GROUP/hpx/issues/1382
[2343] https://github.com/STEllAR-GROUP/hpx/issues/1381
[2344] https://github.com/STEllAR-GROUP/hpx/issues/1380
[2345] https://github.com/STEllAR-GROUP/hpx/issues/1379
[2346] https://github.com/STEllAR-GROUP/hpx/issues/1378
[2347] https://github.com/STEllAR-GROUP/hpx/issues/1377
[2348] https://github.com/STEllAR-GROUP/hpx/issues/1376
[2349] https://github.com/STEllAR-GROUP/hpx/issues/1375

- Issue #1374[2350] - Fixing the MSVC 12 release builder
- Issue #1373[2351] - Modifying parallel search algorithm for zero length searches
- Issue #1372[2352] - Modifying parallel search algorithm for zero length searches
- Issue #1371[2353] - Avoid holding a lock during agas::incref while doing a credit split
- Issue #1370[2354] - `--hpx:bind` throws unexpected error
- Issue #1369[2355] - Getting rid of (void) in loops
- Issue #1368[2356] - Variadic templates support for tuple
- Issue #1367[2357] - One last batch of variadic templates support
- Issue #1366[2358] - Fixing symbolic namespace hang
- Issue #1365[2359] - More held locks
- Issue #1364[2360] - Add counters 1363
- Issue #1363[2361] - Add thread overhead counters
- Issue #1362[2362] - Std config removal
- Issue #1361[2363] - Parcelport plugins
- Issue #1360[2364] - Detuplify transfer_action
- Issue #1359[2365] - Removed obsolete checks
- Issue #1358[2366] - Fixing 1352
- Issue #1357[2367] - Variadic templates support for runtime_support and components
- Issue #1356[2368] - fixed coordinate test for intel13
- Issue #1355[2369] - fixed coordinate.hpp
- Issue #1354[2370] - Lexicographical Compare completed
- Issue #1353[2371] - HPX should set `Boost_ADDITIONAL_VERSIONS` flags
- Issue #1352[2372] - Error: Cannot find action '' in type registry: HPX(bad_action_code)

[2350] https://github.com/STEllAR-GROUP/hpx/issues/1374
[2351] https://github.com/STEllAR-GROUP/hpx/issues/1373
[2352] https://github.com/STEllAR-GROUP/hpx/issues/1372
[2353] https://github.com/STEllAR-GROUP/hpx/issues/1371
[2354] https://github.com/STEllAR-GROUP/hpx/issues/1370
[2355] https://github.com/STEllAR-GROUP/hpx/issues/1369
[2356] https://github.com/STEllAR-GROUP/hpx/issues/1368
[2357] https://github.com/STEllAR-GROUP/hpx/issues/1367
[2358] https://github.com/STEllAR-GROUP/hpx/issues/1366
[2359] https://github.com/STEllAR-GROUP/hpx/issues/1365
[2360] https://github.com/STEllAR-GROUP/hpx/issues/1364
[2361] https://github.com/STEllAR-GROUP/hpx/issues/1363
[2362] https://github.com/STEllAR-GROUP/hpx/issues/1362
[2363] https://github.com/STEllAR-GROUP/hpx/issues/1361
[2364] https://github.com/STEllAR-GROUP/hpx/issues/1360
[2365] https://github.com/STEllAR-GROUP/hpx/issues/1359
[2366] https://github.com/STEllAR-GROUP/hpx/issues/1358
[2367] https://github.com/STEllAR-GROUP/hpx/issues/1357
[2368] https://github.com/STEllAR-GROUP/hpx/issues/1356
[2369] https://github.com/STEllAR-GROUP/hpx/issues/1355
[2370] https://github.com/STEllAR-GROUP/hpx/issues/1354
[2371] https://github.com/STEllAR-GROUP/hpx/issues/1353
[2372] https://github.com/STEllAR-GROUP/hpx/issues/1352

- Issue #1351[2373] - Variadic templates support for appliers
- Issue #1350[2374] - Actions simplification
- Issue #1349[2375] - Variadic when and wait functions
- Issue #1348[2376] - Added hpx_init header to test files
- Issue #1347[2377] - Another batch of variadic templates support
- Issue #1346[2378] - Segmented copy
- Issue #1345[2379] - Attempting to fix hangs during shutdown
- Issue #1344[2380] - Std config removal
- Issue #1343[2381] - Removing various distribution policies for hpx::vector
- Issue #1342[2382] - Inclusive scan
- Issue #1341[2383] - Exclusive scan
- Issue #1340[2384] - Adding `parallel::count` for distributed data structures, adding tests
- Issue #1339[2385] - Update argument order for transform_reduce
- Issue #1337[2386] - Fix dataflow to handle properly ranges of futures
- Issue #1336[2387] - dataflow needs to hold onto futures passed to it
- Issue #1335[2388] - Fails to compile with msvc14
- Issue #1334[2389] - Examples build problem
- Issue #1333[2390] - Distributed transform reduce
- Issue #1332[2391] - Variadic templates support for actions
- Issue #1331[2392] - Some ambiguous calls of map::erase have been prevented by adding additional check in locality constructor.
- Issue #1330[2393] - Defining Plain Actions does not work as described in the documentation
- Issue #1329[2394] - Distributed vector cleanup
- Issue #1328[2395] - Sync docs and comments with code in hello_world example

2373 https://github.com/STEllAR-GROUP/hpx/issues/1351
2374 https://github.com/STEllAR-GROUP/hpx/issues/1350
2375 https://github.com/STEllAR-GROUP/hpx/issues/1349
2376 https://github.com/STEllAR-GROUP/hpx/issues/1348
2377 https://github.com/STEllAR-GROUP/hpx/issues/1347
2378 https://github.com/STEllAR-GROUP/hpx/issues/1346
2379 https://github.com/STEllAR-GROUP/hpx/issues/1345
2380 https://github.com/STEllAR-GROUP/hpx/issues/1344
2381 https://github.com/STEllAR-GROUP/hpx/issues/1343
2382 https://github.com/STEllAR-GROUP/hpx/issues/1342
2383 https://github.com/STEllAR-GROUP/hpx/issues/1341
2384 https://github.com/STEllAR-GROUP/hpx/issues/1340
2385 https://github.com/STEllAR-GROUP/hpx/issues/1339
2386 https://github.com/STEllAR-GROUP/hpx/issues/1337
2387 https://github.com/STEllAR-GROUP/hpx/issues/1336
2388 https://github.com/STEllAR-GROUP/hpx/issues/1335
2389 https://github.com/STEllAR-GROUP/hpx/issues/1334
2390 https://github.com/STEllAR-GROUP/hpx/issues/1333
2391 https://github.com/STEllAR-GROUP/hpx/issues/1332
2392 https://github.com/STEllAR-GROUP/hpx/issues/1331
2393 https://github.com/STEllAR-GROUP/hpx/issues/1330
2394 https://github.com/STEllAR-GROUP/hpx/issues/1329
2395 https://github.com/STEllAR-GROUP/hpx/issues/1328

- Issue #1327[2396] - Typos in docs
- Issue #1326[2397] - Documentation and code diverged in Fibonacci tutorial
- Issue #1325[2398] - Exceptions thrown during parcel handling are not handled correctly
- Issue #1324[2399] - fixed bandwidth calculation
- Issue #1323[2400] - mmap() failed to allocate thread stack due to insufficient resources
- Issue #1322[2401] - HPX fails to build aa182cf
- Issue #1321[2402] - Limiting size of outgoing messages while coalescing parcels
- Issue #1320[2403] - passing a future with launch::deferred in remote function call causes hang
- Issue #1319[2404] - An exception when tries to specify number high priority threads with abp-priority
- Issue #1318[2405] - Unable to run program with abp-priority and numa-sensitivity enabled
- Issue #1317[2406] - N4071 Search/Search_n finished, minor changes
- Issue #1316[2407] - Add config option to make -Ihpx.run_hpx_main!=1 the default
- Issue #1314[2408] - Variadic support for async and apply
- Issue #1313[2409] - Adjust when_any/some to the latest proposed interfaces
- Issue #1312[2410] - Fixing #857: hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #1311[2411] - Distributed get'er/set'er_values for distributed vector
- Issue #1310[2412] - Crashing in hpx::parcelset::policies::mpi::connection_handler::handle_messages() on SuperMIC
- Issue #1308[2413] - Unable to execute an application with –hpx:threads
- Issue #1307[2414] - merge_graph linking issue
- Issue #1306[2415] - First batch of variadic templates support
- Issue #1305[2416] - Create a compiler wrapper
- Issue #1304[2417] - Provide a compiler wrapper for hpx

[2396] https://github.com/STEllAR-GROUP/hpx/issues/1327
[2397] https://github.com/STEllAR-GROUP/hpx/issues/1326
[2398] https://github.com/STEllAR-GROUP/hpx/issues/1325
[2399] https://github.com/STEllAR-GROUP/hpx/issues/1324
[2400] https://github.com/STEllAR-GROUP/hpx/issues/1323
[2401] https://github.com/STEllAR-GROUP/hpx/issues/1322
[2402] https://github.com/STEllAR-GROUP/hpx/issues/1321
[2403] https://github.com/STEllAR-GROUP/hpx/issues/1320
[2404] https://github.com/STEllAR-GROUP/hpx/issues/1319
[2405] https://github.com/STEllAR-GROUP/hpx/issues/1318
[2406] https://github.com/STEllAR-GROUP/hpx/issues/1317
[2407] https://github.com/STEllAR-GROUP/hpx/issues/1316
[2408] https://github.com/STEllAR-GROUP/hpx/issues/1314
[2409] https://github.com/STEllAR-GROUP/hpx/issues/1313
[2410] https://github.com/STEllAR-GROUP/hpx/issues/1312
[2411] https://github.com/STEllAR-GROUP/hpx/issues/1311
[2412] https://github.com/STEllAR-GROUP/hpx/issues/1310
[2413] https://github.com/STEllAR-GROUP/hpx/issues/1308
[2414] https://github.com/STEllAR-GROUP/hpx/issues/1307
[2415] https://github.com/STEllAR-GROUP/hpx/issues/1306
[2416] https://github.com/STEllAR-GROUP/hpx/issues/1305
[2417] https://github.com/STEllAR-GROUP/hpx/issues/1304

- Issue #1303[2418] - Drop support for GCC44
- Issue #1302[2419] - Fixing #1297
- Issue #1301[2420] - Compilation error when tried to use boost range iterators with wait_all
- Issue #1298[2421] - Distributed vector
- Issue #1297[2422] - Unable to invoke component actions recursively
- Issue #1294[2423] - HDF5 build error
- Issue #1275[2424] - The parcelport implementation is non-optimal
- Issue #1267[2425] - Added classes and unit tests for local_file, orangefs_file and pxfs_file
- Issue #1264[2426] - Error "assertion '!m_fun' failed" randomly occurs when using TCP
- Issue #1254[2427] - thread binding seems to not work properly
- Issue #1220[2428] - parallel::copy_if is broken
- Issue #1217[2429] - Find a better way of fixing the issue patched by #1216
- Issue #1168[2430] - Starting HPX on Cray machines using aprun isn't working correctly
- Issue #1085[2431] - Replace startup and shutdown barriers with broadcasts
- Issue #981[2432] - With SLURM, –hpx:threads=8 should not be necessary
- Issue #857[2433] - hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #850[2434] - "flush" not documented
- Issue #763[2435] - Create buildbot instance that uses std::bind as HPX_STD_BIND
- Issue #680[2436] - Convert parcel ports into a plugin system
- Issue #582[2437] - Make exception thrown from HPX threads available from `hpx::init`
- Issue #504[2438] - Refactor Dataflow LCO to work with futures
- Issue #196[2439] - Don't store copies of the locality network metadata in the gva table

[2418] https://github.com/STEllAR-GROUP/hpx/issues/1303
[2419] https://github.com/STEllAR-GROUP/hpx/issues/1302
[2420] https://github.com/STEllAR-GROUP/hpx/issues/1301
[2421] https://github.com/STEllAR-GROUP/hpx/issues/1298
[2422] https://github.com/STEllAR-GROUP/hpx/issues/1297
[2423] https://github.com/STEllAR-GROUP/hpx/issues/1294
[2424] https://github.com/STEllAR-GROUP/hpx/issues/1275
[2425] https://github.com/STEllAR-GROUP/hpx/issues/1267
[2426] https://github.com/STEllAR-GROUP/hpx/issues/1264
[2427] https://github.com/STEllAR-GROUP/hpx/issues/1254
[2428] https://github.com/STEllAR-GROUP/hpx/issues/1220
[2429] https://github.com/STEllAR-GROUP/hpx/issues/1217
[2430] https://github.com/STEllAR-GROUP/hpx/issues/1168
[2431] https://github.com/STEllAR-GROUP/hpx/issues/1085
[2432] https://github.com/STEllAR-GROUP/hpx/issues/981
[2433] https://github.com/STEllAR-GROUP/hpx/issues/857
[2434] https://github.com/STEllAR-GROUP/hpx/issues/850
[2435] https://github.com/STEllAR-GROUP/hpx/issues/763
[2436] https://github.com/STEllAR-GROUP/hpx/issues/680
[2437] https://github.com/STEllAR-GROUP/hpx/issues/582
[2438] https://github.com/STEllAR-GROUP/hpx/issues/504
[2439] https://github.com/STEllAR-GROUP/hpx/issues/196

2.10.9 *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[2440].
- We started to implement various proposals to the C++ Standardization committee related to parallelism and concurrency, most notably N4409[2441] (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411[2442] (Task Region Rev. 3), and N4313[2443] (Working Draft, Technical Specification for C++ Extensions for Concurrency).
- We completely remodeled our automatic build system to run builds and unit tests on various systems and compilers. This allows us to find most bugs right as they were introduced and helps to maintain a high level of quality and compatibility. The newest build logs can be found at *HPX* Buildbot Website[2444].

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1296[2445] - Rename make_error_future to make_exceptional_future, adjust to N4123
- Issue #1295[2446] - building issue
- Issue #1293[2447] - Transpose example

[2440] https://github.com/STEllAR-GROUP/hpx/issues/1204
[2441] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf
[2442] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
[2443] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
[2444] https://rostam.cct.lsu.edu/
[2445] https://github.com/STEllAR-GROUP/hpx/issues/1296
[2446] https://github.com/STEllAR-GROUP/hpx/issues/1295
[2447] https://github.com/STEllAR-GROUP/hpx/issues/1293

- Issue #1292[2448] - Wrong abs() function used in example
- Issue #1291[2449] - non-synchronized shift operators have been removed
- Issue #1290[2450] - RDTSCP is defined as true for Xeon Phi build
- Issue #1289[2451] - Fixing 1288
- Issue #1288[2452] - Add new performance counters
- Issue #1287[2453] - Hierarchy scheduler broken performance counters
- Issue #1286[2454] - Algorithm cleanup
- Issue #1285[2455] - Broken Links in Documentation
- Issue #1284[2456] - Uninitialized copy
- Issue #1283[2457] - missing boost::scoped_ptr includes
- Issue #1282[2458] - Update documentation of build options for schedulers
- Issue #1281[2459] - reset idle rate counter
- Issue #1280[2460] - Bug when executing on Intel MIC
- Issue #1279[2461] - Add improved when_all/wait_all
- Issue #1278[2462] - Implement improved when_all/wait_all
- Issue #1277[2463] - feature request: get access to argc argv and variables_map
- Issue #1276[2464] - Remove merging map
- Issue #1274[2465] - Weird (wrong) string code in papi.cpp
- Issue #1273[2466] - Sequential task execution policy
- Issue #1272[2467] - Avoid CMake name clash for Boost.Thread library
- Issue #1271[2468] - Updates on HPX Test Units
- Issue #1270[2469] - hpx/util/safe_lexical_cast.hpp is added
- Issue #1269[2470] - Added default value for "LIB" cmake variable

2448 https://github.com/STEllAR-GROUP/hpx/issues/1292
2449 https://github.com/STEllAR-GROUP/hpx/issues/1291
2450 https://github.com/STEllAR-GROUP/hpx/issues/1290
2451 https://github.com/STEllAR-GROUP/hpx/issues/1289
2452 https://github.com/STEllAR-GROUP/hpx/issues/1288
2453 https://github.com/STEllAR-GROUP/hpx/issues/1287
2454 https://github.com/STEllAR-GROUP/hpx/issues/1286
2455 https://github.com/STEllAR-GROUP/hpx/issues/1285
2456 https://github.com/STEllAR-GROUP/hpx/issues/1284
2457 https://github.com/STEllAR-GROUP/hpx/issues/1283
2458 https://github.com/STEllAR-GROUP/hpx/issues/1282
2459 https://github.com/STEllAR-GROUP/hpx/issues/1281
2460 https://github.com/STEllAR-GROUP/hpx/issues/1280
2461 https://github.com/STEllAR-GROUP/hpx/issues/1279
2462 https://github.com/STEllAR-GROUP/hpx/issues/1278
2463 https://github.com/STEllAR-GROUP/hpx/issues/1277
2464 https://github.com/STEllAR-GROUP/hpx/issues/1276
2465 https://github.com/STEllAR-GROUP/hpx/issues/1274
2466 https://github.com/STEllAR-GROUP/hpx/issues/1273
2467 https://github.com/STEllAR-GROUP/hpx/issues/1272
2468 https://github.com/STEllAR-GROUP/hpx/issues/1271
2469 https://github.com/STEllAR-GROUP/hpx/issues/1270
2470 https://github.com/STEllAR-GROUP/hpx/issues/1269

- Issue #1268[2471] - Memory Counters not working
- Issue #1266[2472] - FindHPX.cmake is not installed
- Issue #1263[2473] - apply_remote test takes too long
- Issue #1262[2474] - Chrono cleanup
- Issue #1261[2475] - Need make install for papi counters and this builds all the examples
- Issue #1260[2476] - Documentation of Stencil example claims
- Issue #1259[2477] - Avoid double-linking Boost on Windows
- Issue #1257[2478] - Adding additional parameter to create_thread
- Issue #1256[2479] - added buildbot changes to release notes
- Issue #1255[2480] - Cannot build MiniGhost
- Issue #1253[2481] - hpx::thread defects
- Issue #1252[2482] - HPX_PREFIX is too fragile
- Issue #1250[2483] - switch_to_fiber_emulation does not work properly
- Issue #1249[2484] - Documentation is generated under Release folder
- Issue #1248[2485] - Fix usage of hpx_generic_coroutine_context and get tests passing on powerpc
- Issue #1247[2486] - Dynamic linking error
- Issue #1246[2487] - Make cpuid.cpp C++11 compliant
- Issue #1245[2488] - HPX fails on startup (setting thread affinity mask)
- Issue #1244[2489] - HPX_WITH_RDTSC configure test fails, but should succeed
- Issue #1243[2490] - CTest dashboard info for CSCS CDash drop location
- Issue #1242[2491] - Mac fixes
- Issue #1241[2492] - Failure in Distributed with Boost 1.56
- Issue #1240[2493] - fix a race condition in examples.diskperf

2471 https://github.com/STEllAR-GROUP/hpx/issues/1268
2472 https://github.com/STEllAR-GROUP/hpx/issues/1266
2473 https://github.com/STEllAR-GROUP/hpx/issues/1263
2474 https://github.com/STEllAR-GROUP/hpx/issues/1262
2475 https://github.com/STEllAR-GROUP/hpx/issues/1261
2476 https://github.com/STEllAR-GROUP/hpx/issues/1260
2477 https://github.com/STEllAR-GROUP/hpx/issues/1259
2478 https://github.com/STEllAR-GROUP/hpx/issues/1257
2479 https://github.com/STEllAR-GROUP/hpx/issues/1256
2480 https://github.com/STEllAR-GROUP/hpx/issues/1255
2481 https://github.com/STEllAR-GROUP/hpx/issues/1253
2482 https://github.com/STEllAR-GROUP/hpx/issues/1252
2483 https://github.com/STEllAR-GROUP/hpx/issues/1250
2484 https://github.com/STEllAR-GROUP/hpx/issues/1249
2485 https://github.com/STEllAR-GROUP/hpx/issues/1248
2486 https://github.com/STEllAR-GROUP/hpx/issues/1247
2487 https://github.com/STEllAR-GROUP/hpx/issues/1246
2488 https://github.com/STEllAR-GROUP/hpx/issues/1245
2489 https://github.com/STEllAR-GROUP/hpx/issues/1244
2490 https://github.com/STEllAR-GROUP/hpx/issues/1243
2491 https://github.com/STEllAR-GROUP/hpx/issues/1242
2492 https://github.com/STEllAR-GROUP/hpx/issues/1241
2493 https://github.com/STEllAR-GROUP/hpx/issues/1240

- Issue #1239[2494] - fix wait_each in examples.diskperf
- Issue #1238[2495] - Fixed #1237: hpx::util::portable_binary_iarchive failed
- Issue #1237[2496] - hpx::util::portable_binary_iarchive faileds
- Issue #1235[2497] - Fixing clang warnings and errors
- Issue #1234[2498] - TCP runs fail: Transport endpoint is not connected
- Issue #1233[2499] - Making sure the correct number of threads is registered with AGAS
- Issue #1232[2500] - Fixing race in wait_xxx
- Issue #1231[2501] - Parallel minmax
- Issue #1230[2502] - Distributed run of 1d_stencil_8 uses less threads than spec. & sometimes gives errors
- Issue #1229[2503] - Unstable number of threads
- Issue #1228[2504] - HPX link error (cmake / MPI)
- Issue #1226[2505] - Warning about struct/class thread_counters
- Issue #1225[2506] - Adding parallel::replace etc
- Issue #1224[2507] - Extending dataflow to pass through non-future arguments
- Issue #1223[2508] - Remaining find algorithms implemented, N4071
- Issue #1222[2509] - Merging all the changes
- Issue #1221[2510] - No error output when using mpirun with hpx
- Issue #1219[2511] - Adding new AGAS cache performance counters
- Issue #1216[2512] - Fixing using futures (clients) as arguments to actions
- Issue #1215[2513] - Error compiling simple component
- Issue #1214[2514] - Stencil docs
- Issue #1213[2515] - Using more than a few dozen MPI processes on SuperMike results in a seg fault before getting to hpx_main
- Issue #1212[2516] - Parallel rotate

2494 https://github.com/STEllAR-GROUP/hpx/issues/1239
2495 https://github.com/STEllAR-GROUP/hpx/issues/1238
2496 https://github.com/STEllAR-GROUP/hpx/issues/1237
2497 https://github.com/STEllAR-GROUP/hpx/issues/1235
2498 https://github.com/STEllAR-GROUP/hpx/issues/1234
2499 https://github.com/STEllAR-GROUP/hpx/issues/1233
2500 https://github.com/STEllAR-GROUP/hpx/issues/1232
2501 https://github.com/STEllAR-GROUP/hpx/issues/1231
2502 https://github.com/STEllAR-GROUP/hpx/issues/1230
2503 https://github.com/STEllAR-GROUP/hpx/issues/1229
2504 https://github.com/STEllAR-GROUP/hpx/issues/1228
2505 https://github.com/STEllAR-GROUP/hpx/issues/1226
2506 https://github.com/STEllAR-GROUP/hpx/issues/1225
2507 https://github.com/STEllAR-GROUP/hpx/issues/1224
2508 https://github.com/STEllAR-GROUP/hpx/issues/1223
2509 https://github.com/STEllAR-GROUP/hpx/issues/1222
2510 https://github.com/STEllAR-GROUP/hpx/issues/1221
2511 https://github.com/STEllAR-GROUP/hpx/issues/1219
2512 https://github.com/STEllAR-GROUP/hpx/issues/1216
2513 https://github.com/STEllAR-GROUP/hpx/issues/1215
2514 https://github.com/STEllAR-GROUP/hpx/issues/1214
2515 https://github.com/STEllAR-GROUP/hpx/issues/1213
2516 https://github.com/STEllAR-GROUP/hpx/issues/1212

- Issue #1211[2517] - Direct actions cause the future's shared_state to be leaked
- Issue #1210[2518] - Refactored local::promise to be standard conformant
- Issue #1209[2519] - Improve command line handling
- Issue #1208[2520] - Adding parallel::reverse and parallel::reverse_copy
- Issue #1207[2521] - Add copy_backward and move_backward
- Issue #1206[2522] - N4071 additional algorithms implemented
- Issue #1204[2523] - Cmake simplification and various other minor changes
- Issue #1203[2524] - Implementing new launch policy for (local) async: `hpx::launch::fork`.
- Issue #1202[2525] - Failed assertion in connection_cache.hpp
- Issue #1201[2526] - pkg-config doesn't add mpi link directories
- Issue #1200[2527] - Error when querying time performance counters
- Issue #1199[2528] - library path is now configurable (again)
- Issue #1198[2529] - Error when querying performance counters
- Issue #1197[2530] - tests fail with intel compiler
- Issue #1196[2531] - Silence several warnings
- Issue #1195[2532] - Rephrase initializers to work with VC++ 2012
- Issue #1194[2533] - Simplify parallel algorithms
- Issue #1193[2534] - Adding `parallel::equal`
- Issue #1192[2535] - HPX(out_of_memory) on including <hpx/hpx.hpp>
- Issue #1191[2536] - Fixing #1189
- Issue #1190[2537] - Chrono cleanup
- Issue #1189[2538] - Deadlock .. somewhere? (probably serialization)
- Issue #1188[2539] - Removed `future::get_status()`

2517 https://github.com/STEllAR-GROUP/hpx/issues/1211
2518 https://github.com/STEllAR-GROUP/hpx/issues/1210
2519 https://github.com/STEllAR-GROUP/hpx/issues/1209
2520 https://github.com/STEllAR-GROUP/hpx/issues/1208
2521 https://github.com/STEllAR-GROUP/hpx/issues/1207
2522 https://github.com/STEllAR-GROUP/hpx/issues/1206
2523 https://github.com/STEllAR-GROUP/hpx/issues/1204
2524 https://github.com/STEllAR-GROUP/hpx/issues/1203
2525 https://github.com/STEllAR-GROUP/hpx/issues/1202
2526 https://github.com/STEllAR-GROUP/hpx/issues/1201
2527 https://github.com/STEllAR-GROUP/hpx/issues/1200
2528 https://github.com/STEllAR-GROUP/hpx/issues/1199
2529 https://github.com/STEllAR-GROUP/hpx/issues/1198
2530 https://github.com/STEllAR-GROUP/hpx/issues/1197
2531 https://github.com/STEllAR-GROUP/hpx/issues/1196
2532 https://github.com/STEllAR-GROUP/hpx/issues/1195
2533 https://github.com/STEllAR-GROUP/hpx/issues/1194
2534 https://github.com/STEllAR-GROUP/hpx/issues/1193
2535 https://github.com/STEllAR-GROUP/hpx/issues/1192
2536 https://github.com/STEllAR-GROUP/hpx/issues/1191
2537 https://github.com/STEllAR-GROUP/hpx/issues/1190
2538 https://github.com/STEllAR-GROUP/hpx/issues/1189
2539 https://github.com/STEllAR-GROUP/hpx/issues/1188

- Issue #1186[2540] - Fixed FindOpenCL to find current AMD APP SDK
- Issue #1184[2541] - Tweaking future unwrapping
- Issue #1183[2542] - Extended `parallel::reduce`
- Issue #1182[2543] - `future::unwrap` hangs for `launch::deferred`
- Issue #1181[2544] - Adding `all_of`, `any_of`, and `none_of` and corresponding documentation
- Issue #1180[2545] - `hpx::cout` defect
- Issue #1179[2546] - `hpx::async` does not work for member function pointers when called on types with self-defined unary `operator*`
- Issue #1178[2547] - Implemented variadic `hpx::util::zip_iterator`
- Issue #1177[2548] - MPI parcelport defect
- Issue #1176[2549] - `HPX_DEFINE_COMPONENT_CONST_ACTION_TPL` does not have a 2-argument version
- Issue #1175[2550] - Create util::zip_iterator working with util::tuple<>
- Issue #1174[2551] - Error Building HPX on linux, root_certificate_authority.cpp
- Issue #1173[2552] - hpx::cout output lost
- Issue #1172[2553] - HPX build error with Clang 3.4.2
- Issue #1171[2554] - `CMAKE_INSTALL_PREFIX` ignored
- Issue #1170[2555] - Close hpx_benchmarks repository on Github
- Issue #1169[2556] - Buildbot emails have syntax error in url
- Issue #1167[2557] - Merge partial implementation of standards proposal N3960
- Issue #1166[2558] - Fixed several compiler warnings
- Issue #1165[2559] - cmake warns: "tests.regressions.actions" does not exist
- Issue #1164[2560] - Want my own serialization of hpx::future
- Issue #1162[2561] - Segfault in hello_world example
- Issue #1161[2562] - Use `HPX_ASSERT` to aid the compiler

2540 https://github.com/STEllAR-GROUP/hpx/issues/1186
2541 https://github.com/STEllAR-GROUP/hpx/issues/1184
2542 https://github.com/STEllAR-GROUP/hpx/issues/1183
2543 https://github.com/STEllAR-GROUP/hpx/issues/1182
2544 https://github.com/STEllAR-GROUP/hpx/issues/1181
2545 https://github.com/STEllAR-GROUP/hpx/issues/1180
2546 https://github.com/STEllAR-GROUP/hpx/issues/1179
2547 https://github.com/STEllAR-GROUP/hpx/issues/1178
2548 https://github.com/STEllAR-GROUP/hpx/issues/1177
2549 https://github.com/STEllAR-GROUP/hpx/issues/1176
2550 https://github.com/STEllAR-GROUP/hpx/issues/1175
2551 https://github.com/STEllAR-GROUP/hpx/issues/1174
2552 https://github.com/STEllAR-GROUP/hpx/issues/1173
2553 https://github.com/STEllAR-GROUP/hpx/issues/1172
2554 https://github.com/STEllAR-GROUP/hpx/issues/1171
2555 https://github.com/STEllAR-GROUP/hpx/issues/1170
2556 https://github.com/STEllAR-GROUP/hpx/issues/1169
2557 https://github.com/STEllAR-GROUP/hpx/issues/1167
2558 https://github.com/STEllAR-GROUP/hpx/issues/1166
2559 https://github.com/STEllAR-GROUP/hpx/issues/1165
2560 https://github.com/STEllAR-GROUP/hpx/issues/1164
2561 https://github.com/STEllAR-GROUP/hpx/issues/1162
2562 https://github.com/STEllAR-GROUP/hpx/issues/1161

- Issue #1160[2563] - Do not put -DNDEBUG into hpx_application.pc
- Issue #1159[2564] - Support Clang 3.4.2
- Issue #1158[2565] - Fixed #1157: Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1157[2566] - Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1156[2567] - Force inlining fails
- Issue #1155[2568] - changed header of printout to be compatible with python csv module
- Issue #1154[2569] - Fixing iostreams
- Issue #1153[2570] - Standard manipulators (like std::endl) do not work with hpx::ostream
- Issue #1152[2571] - Functions revamp
- Issue #1151[2572] - Supressing cmake 3.0 policy warning for CMP0026
- Issue #1150[2573] - Client Serialization error
- Issue #1149[2574] - Segfault on Stampede
- Issue #1148[2575] - Refactoring mini-ghost
- Issue #1147[2576] - N3960 copy_if and copy_n implemented and tested
- Issue #1146[2577] - Stencil print
- Issue #1145[2578] - N3960 hpx::parallel::copy implemented and tested
- Issue #1144[2579] - OpenMP examples 1d_stencil do not build
- Issue #1143[2580] - 1d_stencil OpenMP examples do not build
- Issue #1142[2581] - Cannot build HPX with gcc 4.6 on OS X
- Issue #1140[2582] - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
- Issue #1139[2583] - hpx/hpx/config/compiler_specific.hpp
- Issue #1138[2584] - clean up pkg-config files
- Issue #1137[2585] - Improvements to create binary packages

2563 https://github.com/STEllAR-GROUP/hpx/issues/1160
2564 https://github.com/STEllAR-GROUP/hpx/issues/1159
2565 https://github.com/STEllAR-GROUP/hpx/issues/1158
2566 https://github.com/STEllAR-GROUP/hpx/issues/1157
2567 https://github.com/STEllAR-GROUP/hpx/issues/1156
2568 https://github.com/STEllAR-GROUP/hpx/issues/1155
2569 https://github.com/STEllAR-GROUP/hpx/issues/1154
2570 https://github.com/STEllAR-GROUP/hpx/issues/1153
2571 https://github.com/STEllAR-GROUP/hpx/issues/1152
2572 https://github.com/STEllAR-GROUP/hpx/issues/1151
2573 https://github.com/STEllAR-GROUP/hpx/issues/1150
2574 https://github.com/STEllAR-GROUP/hpx/issues/1149
2575 https://github.com/STEllAR-GROUP/hpx/issues/1148
2576 https://github.com/STEllAR-GROUP/hpx/issues/1147
2577 https://github.com/STEllAR-GROUP/hpx/issues/1146
2578 https://github.com/STEllAR-GROUP/hpx/issues/1145
2579 https://github.com/STEllAR-GROUP/hpx/issues/1144
2580 https://github.com/STEllAR-GROUP/hpx/issues/1143
2581 https://github.com/STEllAR-GROUP/hpx/issues/1142
2582 https://github.com/STEllAR-GROUP/hpx/issues/1140
2583 https://github.com/STEllAR-GROUP/hpx/issues/1139
2584 https://github.com/STEllAR-GROUP/hpx/issues/1138
2585 https://github.com/STEllAR-GROUP/hpx/issues/1137

- Issue #1136[2586] - HPX_GCC_VERSION not defined on all compilers
- Issue #1135[2587] - Avoiding collision between winsock2.h and windows.h
- Issue #1134[2588] - Making sure, that hpx::finalize can be called from any locality
- Issue #1133[2589] - 1d stencil examples
- Issue #1131[2590] - Refactor unique_function implementation
- Issue #1130[2591] - Unique function
- Issue #1129[2592] - Some fixes to the Build system on OS X
- Issue #1128[2593] - Action future args
- Issue #1127[2594] - Executor causes segmentation fault
- Issue #1124[2595] - Adding new API functions: `register_id_with_basename`, `unregister_id_with_basename`, `find_ids_from_basename`; adding test
- Issue #1123[2596] - Reduce nesting of try-catch construct in `encode_parcels`?
- Issue #1122[2597] - Client base fixes
- Issue #1121[2598] - Update `hpxrun.py.in`
- Issue #1120[2599] - HTTS2 tests compile errors on v110 (VS2012)
- Issue #1119[2600] - Remove references to boost::atomic in accumulator example
- Issue #1118[2601] - Only build test thread_pool_executor_1114_test if `HPX_LOCAL_SCHEDULER` is set
- Issue #1117[2602] - local_queue_executor linker error on vc110
- Issue #1116[2603] - Disabled performance counter should give runtime errors, not invalid data
- Issue #1115[2604] - Compile error with Intel C++ 13.1
- Issue #1114[2605] - Default constructed executor is not usable
- Issue #1113[2606] - Fast compilation of logging causes ABI incompatibilities between different `NDEBUG` values
- Issue #1112[2607] - Using thread_pool_executors causes segfault
- Issue #1111[2608] - `hpx::threads::get_thread_data` always returns zero

[2586] https://github.com/STEllAR-GROUP/hpx/issues/1136
[2587] https://github.com/STEllAR-GROUP/hpx/issues/1135
[2588] https://github.com/STEllAR-GROUP/hpx/issues/1134
[2589] https://github.com/STEllAR-GROUP/hpx/issues/1133
[2590] https://github.com/STEllAR-GROUP/hpx/issues/1131
[2591] https://github.com/STEllAR-GROUP/hpx/issues/1130
[2592] https://github.com/STEllAR-GROUP/hpx/issues/1129
[2593] https://github.com/STEllAR-GROUP/hpx/issues/1128
[2594] https://github.com/STEllAR-GROUP/hpx/issues/1127
[2595] https://github.com/STEllAR-GROUP/hpx/issues/1124
[2596] https://github.com/STEllAR-GROUP/hpx/issues/1123
[2597] https://github.com/STEllAR-GROUP/hpx/issues/1122
[2598] https://github.com/STEllAR-GROUP/hpx/issues/1121
[2599] https://github.com/STEllAR-GROUP/hpx/issues/1120
[2600] https://github.com/STEllAR-GROUP/hpx/issues/1119
[2601] https://github.com/STEllAR-GROUP/hpx/issues/1118
[2602] https://github.com/STEllAR-GROUP/hpx/issues/1117
[2603] https://github.com/STEllAR-GROUP/hpx/issues/1116
[2604] https://github.com/STEllAR-GROUP/hpx/issues/1115
[2605] https://github.com/STEllAR-GROUP/hpx/issues/1114
[2606] https://github.com/STEllAR-GROUP/hpx/issues/1113
[2607] https://github.com/STEllAR-GROUP/hpx/issues/1112
[2608] https://github.com/STEllAR-GROUP/hpx/issues/1111

- Issue #1110[2609] - Remove unnecessary null pointer checks
- Issue #1109[2610] - More tests adjustments
- Issue #1108[2611] - Clarify build rules for "libboost_atomic-mt.so"?
- Issue #1107[2612] - Remove unnecessary null pointer checks
- Issue #1106[2613] - network_storage benchmark imporvements, adding legends to plots and tidying layout
- Issue #1105[2614] - Add more plot outputs and improve instructions doc
- Issue #1104[2615] - Complete quoting for parameters of some CMake commands
- Issue #1103[2616] - Work on test/scripts
- Issue #1102[2617] - Changed minimum requirement of window install to 2012
- Issue #1101[2618] - Changed minimum requirement of window install to 2012
- Issue #1100[2619] - Changed readme to no longer specify using MSVC 2010 compiler
- Issue #1099[2620] - Error returning futures from component actions
- Issue #1098[2621] - Improve storage test
- Issue #1097[2622] - data_actions quickstart example calls missing function decorate_action of data_get_action
- Issue #1096[2623] - MPI parcelport broken with new zero copy optimization
- Issue #1095[2624] - Warning C4005: _WIN32_WINNT: Macro redefinition
- Issue #1094[2625] - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
- Issue #1093[2626] - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS
- Issue #1092[2627] - Rename unique_future<> back to future<>
- Issue #1091[2628] - Inconsistent error message
- Issue #1090[2629] - On windows 8.1 the examples crashed if using more than one os thread
- Issue #1089[2630] - Components should be allowed to have their own executor
- Issue #1088[2631] - Add possibility to select a network interface for the ibverbs parcelport

2609 https://github.com/STEllAR-GROUP/hpx/issues/1110
2610 https://github.com/STEllAR-GROUP/hpx/issues/1109
2611 https://github.com/STEllAR-GROUP/hpx/issues/1108
2612 https://github.com/STEllAR-GROUP/hpx/issues/1107
2613 https://github.com/STEllAR-GROUP/hpx/issues/1106
2614 https://github.com/STEllAR-GROUP/hpx/issues/1105
2615 https://github.com/STEllAR-GROUP/hpx/issues/1104
2616 https://github.com/STEllAR-GROUP/hpx/issues/1103
2617 https://github.com/STEllAR-GROUP/hpx/issues/1102
2618 https://github.com/STEllAR-GROUP/hpx/issues/1101
2619 https://github.com/STEllAR-GROUP/hpx/issues/1100
2620 https://github.com/STEllAR-GROUP/hpx/issues/1099
2621 https://github.com/STEllAR-GROUP/hpx/issues/1098
2622 https://github.com/STEllAR-GROUP/hpx/issues/1097
2623 https://github.com/STEllAR-GROUP/hpx/issues/1096
2624 https://github.com/STEllAR-GROUP/hpx/issues/1095
2625 https://github.com/STEllAR-GROUP/hpx/issues/1094
2626 https://github.com/STEllAR-GROUP/hpx/issues/1093
2627 https://github.com/STEllAR-GROUP/hpx/issues/1092
2628 https://github.com/STEllAR-GROUP/hpx/issues/1091
2629 https://github.com/STEllAR-GROUP/hpx/issues/1090
2630 https://github.com/STEllAR-GROUP/hpx/issues/1089
2631 https://github.com/STEllAR-GROUP/hpx/issues/1088

- Issue #1087[2632] - ibverbs and ipc parcelport uses zero copy optimization
- Issue #1083[2633] - Make shell examples copyable in docs
- Issue #1082[2634] - Implement proper termination detection during shutdown
- Issue #1081[2635] - Implement thread_specific_ptr for hpx::threads
- Issue #1072[2636] - make install not working properly
- Issue #1070[2637] - Complete quoting for parameters of some CMake commands
- Issue #1059[2638] - Fix more unused variable warnings
- Issue #1051[2639] - Implement when_each
- Issue #973[2640] - Would like option to report hwloc bindings
- Issue #970[2641] - Bad flags for Fortran compiler
- Issue #941[2642] - Create a proper user level context switching class for BG/Q
- Issue #935[2643] - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #934[2644] - Want to build HPX without dynamic libraries
- Issue #927[2645] - Make hpx/lcos/reduce.hpp accept futures of id_type
- Issue #926[2646] - All unit tests that are run with more than one thread with CTest/hpx_run_test should configure hpx.os_threads
- Issue #925[2647] - regression_dataflow_791 needs to be brought in line with HPX standards
- Issue #899[2648] - Fix race conditions in regression tests
- Issue #879[2649] - Hung test leads to cascading test failure; make tests should support the MPI parcelport
- Issue #865[2650] - future<T> and friends shall work for movable only Ts
- Issue #847[2651] - Dynamic libraries are not installed on OS X
- Issue #816[2652] - First Program tutorial pull request
- Issue #799[2653] - Wrap lexical_cast to avoid exceptions
- Issue #720[2654] - broken configuration when using ccmake on Ubuntu

[2632] https://github.com/STEllAR-GROUP/hpx/issues/1087
[2633] https://github.com/STEllAR-GROUP/hpx/issues/1083
[2634] https://github.com/STEllAR-GROUP/hpx/issues/1082
[2635] https://github.com/STEllAR-GROUP/hpx/issues/1081
[2636] https://github.com/STEllAR-GROUP/hpx/issues/1072
[2637] https://github.com/STEllAR-GROUP/hpx/issues/1070
[2638] https://github.com/STEllAR-GROUP/hpx/issues/1059
[2639] https://github.com/STEllAR-GROUP/hpx/issues/1051
[2640] https://github.com/STEllAR-GROUP/hpx/issues/973
[2641] https://github.com/STEllAR-GROUP/hpx/issues/970
[2642] https://github.com/STEllAR-GROUP/hpx/issues/941
[2643] https://github.com/STEllAR-GROUP/hpx/issues/935
[2644] https://github.com/STEllAR-GROUP/hpx/issues/934
[2645] https://github.com/STEllAR-GROUP/hpx/issues/927
[2646] https://github.com/STEllAR-GROUP/hpx/issues/926
[2647] https://github.com/STEllAR-GROUP/hpx/issues/925
[2648] https://github.com/STEllAR-GROUP/hpx/issues/899
[2649] https://github.com/STEllAR-GROUP/hpx/issues/879
[2650] https://github.com/STEllAR-GROUP/hpx/issues/865
[2651] https://github.com/STEllAR-GROUP/hpx/issues/847
[2652] https://github.com/STEllAR-GROUP/hpx/issues/816
[2653] https://github.com/STEllAR-GROUP/hpx/issues/799
[2654] https://github.com/STEllAR-GROUP/hpx/issues/720

- Issue #622[2655] - `--hpx:hpx` and `--hpx:debug-hpx-log` is nonsensical
- Issue #525[2656] - Extend barrier LCO test to run in distributed
- Issue #515[2657] - Multi-destination version of hpx::apply is broken
- Issue #509[2658] - Push Boost.Atomic changes upstream
- Issue #503[2659] - Running HPX applications on Windows should not require setting %PATH%
- Issue #461[2660] - Add a compilation sanity test
- Issue #456[2661] - hpx_run_tests.py should log output from tests that timeout
- Issue #454[2662] - Investigate threadmanager performance
- Issue #345[2663] - Add more versatile environmental/cmake variable support to hpx_find_* CMake macros
- Issue #209[2664] - Support multiple configurations in generated build files
- Issue #190[2665] - hpx::cout should be a std::ostream
- Issue #189[2666] - iostreams component should use startup/shutdown functions
- Issue #183[2667] - Use Boost.ICL for correctness in AGAS
- Issue #44[2668] - Implement real futures

2.10.10 *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 todays 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.

[2655] https://github.com/STEllAR-GROUP/hpx/issues/622
[2656] https://github.com/STEllAR-GROUP/hpx/issues/525
[2657] https://github.com/STEllAR-GROUP/hpx/issues/515
[2658] https://github.com/STEllAR-GROUP/hpx/issues/509
[2659] https://github.com/STEllAR-GROUP/hpx/issues/503
[2660] https://github.com/STEllAR-GROUP/hpx/issues/461
[2661] https://github.com/STEllAR-GROUP/hpx/issues/456
[2662] https://github.com/STEllAR-GROUP/hpx/issues/454
[2663] https://github.com/STEllAR-GROUP/hpx/issues/345
[2664] https://github.com/STEllAR-GROUP/hpx/issues/209
[2665] https://github.com/STEllAR-GROUP/hpx/issues/190
[2666] https://github.com/STEllAR-GROUP/hpx/issues/189
[2667] https://github.com/STEllAR-GROUP/hpx/issues/183
[2668] https://github.com/STEllAR-GROUP/hpx/issues/44

- 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[2669]. 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[2670]. 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[2671].
- A large part of the code base of *HPX* has been refactored and partially re-implemented. The main changes were related to
  - The threading subsystem: these changes significantly reduce the amount of overheads caused by the schedulers, improve the modularity of the code base, and extend the variety of available scheduling algorithms.
  - The parcel subsystem: these changes improve the performance of the *HPX* networking layer, modularize the structure of the parcelports, and simplify the creation of new parcelports for other underlying networking libraries.
  - The API subsystem: these changes improved the conformance of the API to C++11 Standard, extend and unify the available API functionality, and decrease the overheads created by various elements of the API.
  - The robustness of the component loading subsystem has been improved significantly, allowing to more portably and more reliably register the components needed by an application as startup. This additionally speeds up general application initialization.
- We added new API functionality like `hpx::migrate` and `hpx::copy_component` which are the basic building blocks necessary for implementing higher level abstractions for system-wide load balancing, runtime-adaptive resource management, and object-oriented checkpointing and state-management.
- We removed the use of C++11 move emulation (using Boost.Move), replacing it with C++11 rvalue references. This is the first step towards using more and more native C++11 facilities which we plan to introduce in the future.
- We improved the reference counting scheme used by *HPX* which helps managing distributed objects and memory. This improves the overall stability of *HPX* and further simplifies writing real world applications.
- The minimal Boost version required to use *HPX* is now V1.49.0.
- This release coincides with the first release of HPXPI (V0.1.0), the first implementation of the XPI specification[2672].

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

[2669] http://www.open-std.org/jtc1/sc22/wg21
[2670] http://mail.cct.lsu.edu/pipermail/hpx-users/2014-January/000141.html
[2671] http://www.open-std.org/jtc1/sc22/wg21
[2672] https://github.com/STEllAR-GROUP/hpxpi/blob/master/spec.pdf?raw=true

- Issue #1086[2673] - Expose internal boost::shared_array to allow user management of array lifetime
- Issue #1083[2674] - Make shell examples copyable in docs
- Issue #1080[2675] - /threads{locality#*/total}/count/cumulative broken
- Issue #1079[2676] - Build problems on OS X
- Issue #1078[2677] - Improve robustness of component loading
- Issue #1077[2678] - Fix a missing enum definition for 'take' mode
- Issue #1076[2679] - Merge Jb master
- Issue #1075[2680] - Unknown CMake command "add_hpx_pseudo_target"
- Issue #1074[2681] - Implement `apply_continue_callback` and `apply_colocated_callback`
- Issue #1073[2682] - The new `apply_colocated` and `async_colocated` functions lead to automatic registered functions
- Issue #1071[2683] - Remove deferred_packaged_task
- Issue #1069[2684] - serialize_buffer with allocator fails at destruction
- Issue #1068[2685] - Coroutine include and forward declarations missing
- Issue #1067[2686] - Add allocator support to util::serialize_buffer
- Issue #1066[2687] - Allow for MPI_Init being called before HPX launches
- Issue #1065[2688] - AGAS cache isn't used/populated on worker localities
- Issue #1064[2689] - Reorder includes to ensure ws2 includes early
- Issue #1063[2690] - Add `hpx::runtime::suspend` and `hpx::runtime::resume`
- Issue #1062[2691] - Fix `async_continue` to propery handle return types
- Issue #1061[2692] - Implement `async_colocated` and `apply_colocated`
- Issue #1060[2693] - Implement minimal component migration
- Issue #1058[2694] - Remove `HPX_UTIL_TUPLE` from code base
- Issue #1057[2695] - Add performance counters for threading subsystem

2673 https://github.com/STEllAR-GROUP/hpx/issues/1086
2674 https://github.com/STEllAR-GROUP/hpx/issues/1083
2675 https://github.com/STEllAR-GROUP/hpx/issues/1080
2676 https://github.com/STEllAR-GROUP/hpx/issues/1079
2677 https://github.com/STEllAR-GROUP/hpx/issues/1078
2678 https://github.com/STEllAR-GROUP/hpx/issues/1077
2679 https://github.com/STEllAR-GROUP/hpx/issues/1076
2680 https://github.com/STEllAR-GROUP/hpx/issues/1075
2681 https://github.com/STEllAR-GROUP/hpx/issues/1074
2682 https://github.com/STEllAR-GROUP/hpx/issues/1073
2683 https://github.com/STEllAR-GROUP/hpx/issues/1071
2684 https://github.com/STEllAR-GROUP/hpx/issues/1069
2685 https://github.com/STEllAR-GROUP/hpx/issues/1068
2686 https://github.com/STEllAR-GROUP/hpx/issues/1067
2687 https://github.com/STEllAR-GROUP/hpx/issues/1066
2688 https://github.com/STEllAR-GROUP/hpx/issues/1065
2689 https://github.com/STEllAR-GROUP/hpx/issues/1064
2690 https://github.com/STEllAR-GROUP/hpx/issues/1063
2691 https://github.com/STEllAR-GROUP/hpx/issues/1062
2692 https://github.com/STEllAR-GROUP/hpx/issues/1061
2693 https://github.com/STEllAR-GROUP/hpx/issues/1060
2694 https://github.com/STEllAR-GROUP/hpx/issues/1058
2695 https://github.com/STEllAR-GROUP/hpx/issues/1057

- Issue #1055[2696] - Thread allocation uses two memory pools
- Issue #1053[2697] - Work stealing flawed
- Issue #1052[2698] - Fix a number of warnings
- Issue #1049[2699] - Fixes for TLS on OSX and more reliable test running
- Issue #1048[2700] - Fixing after 588 hang
- Issue #1047[2701] - Use port '0' for networking when using one locality
- Issue #1046[2702] - `composable_guard` test is broken when having more than one thread
- Issue #1045[2703] - Security missing headers
- Issue #1044[2704] - Native TLS on FreeBSD via __thread
- Issue #1043[2705] - `async` et.al. compute the wrong result type
- Issue #1042[2706] - `async` et.al. implicitly unwrap reference_wrappers
- Issue #1041[2707] - Remove redundant costly Kleene stars from regex searches
- Issue #1040[2708] - CMake script regex match patterns has unnecessary kleenes
- Issue #1039[2709] - Remove use of Boost.Move and replace with std::move and real rvalue refs
- Issue #1038[2710] - Bump minimal required Boost to 1.49.0
- Issue #1037[2711] - Implicit unwrapping of futures in async broken
- Issue #1036[2712] - Scheduler hangs when user code attempts to "block" OS-threads
- Issue #1035[2713] - Idle-rate counter always reports 100% idle rate
- Issue #1034[2714] - Symbolic name registration causes application hangs
- Issue #1033[2715] - Application options read in from an options file generate an error message
- Issue #1032[2716] - `hpx::id_type` local reference counting is wrong
- Issue #1031[2717] - Negative entry in reference count table
- Issue #1030[2718] - Implement condition_variable

[2696] https://github.com/STEllAR-GROUP/hpx/issues/1055
[2697] https://github.com/STEllAR-GROUP/hpx/issues/1053
[2698] https://github.com/STEllAR-GROUP/hpx/issues/1052
[2699] https://github.com/STEllAR-GROUP/hpx/issues/1049
[2700] https://github.com/STEllAR-GROUP/hpx/issues/1048
[2701] https://github.com/STEllAR-GROUP/hpx/issues/1047
[2702] https://github.com/STEllAR-GROUP/hpx/issues/1046
[2703] https://github.com/STEllAR-GROUP/hpx/issues/1045
[2704] https://github.com/STEllAR-GROUP/hpx/issues/1044
[2705] https://github.com/STEllAR-GROUP/hpx/issues/1043
[2706] https://github.com/STEllAR-GROUP/hpx/issues/1042
[2707] https://github.com/STEllAR-GROUP/hpx/issues/1041
[2708] https://github.com/STEllAR-GROUP/hpx/issues/1040
[2709] https://github.com/STEllAR-GROUP/hpx/issues/1039
[2710] https://github.com/STEllAR-GROUP/hpx/issues/1038
[2711] https://github.com/STEllAR-GROUP/hpx/issues/1037
[2712] https://github.com/STEllAR-GROUP/hpx/issues/1036
[2713] https://github.com/STEllAR-GROUP/hpx/issues/1035
[2714] https://github.com/STEllAR-GROUP/hpx/issues/1034
[2715] https://github.com/STEllAR-GROUP/hpx/issues/1033
[2716] https://github.com/STEllAR-GROUP/hpx/issues/1032
[2717] https://github.com/STEllAR-GROUP/hpx/issues/1031
[2718] https://github.com/STEllAR-GROUP/hpx/issues/1030

- Issue #1029[2719] - Deadlock in thread scheduling subsystem
- Issue #1028[2720] - HPX-thread cumulative count performance counters report incorrect value
- Issue #1027[2721] - Expose `hpx::thread_interrupted` error code as a separate exception type
- Issue #1026[2722] - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
- Issue #1025[2723] - `future::wait_for/wait_until` do not remove callback
- Issue #1024[2724] - Remove dependence to boost assert and create hpx assert
- Issue #1023[2725] - Segfaults with tcmalloc
- Issue #1022[2726] - prerequisites link in readme is broken
- Issue #1020[2727] - HPX Deadlock on external synchronization
- Issue #1019[2728] - Convert using `BOOST_ASSERT` to `HPX_ASSERT`
- Issue #1018[2729] - compiling bug with gcc 4.8.1
- Issue #1017[2730] - Possible crash in io_pool executor
- Issue #1016[2731] - Crash at startup
- Issue #1014[2732] - Implement Increment/Decrement Merging
- Issue #1013[2733] - Add more logging channels to enable greater control over logging granularity
- Issue #1012[2734] - `--hpx:debug-hpx-log` and `--hpx:debug-agas-log` lead to non-thread safe writes
- Issue #1011[2735] - After installation, running applications from the build/staging directory no longer works
- Issue #1010[2736] - Mergable decrement requests are not being merged
- Issue #1009[2737] - `--hpx:list-symbolic-names` crashes
- Issue #1007[2738] - Components are not properly destroyed
- Issue #1006[2739] - Segfault/hang in set_data
- Issue #1003[2740] - Performance counter naming issue
- Issue #982[2741] - Race condition during startup

2719 https://github.com/STEllAR-GROUP/hpx/issues/1029
2720 https://github.com/STEllAR-GROUP/hpx/issues/1028
2721 https://github.com/STEllAR-GROUP/hpx/issues/1027
2722 https://github.com/STEllAR-GROUP/hpx/issues/1026
2723 https://github.com/STEllAR-GROUP/hpx/issues/1025
2724 https://github.com/STEllAR-GROUP/hpx/issues/1024
2725 https://github.com/STEllAR-GROUP/hpx/issues/1023
2726 https://github.com/STEllAR-GROUP/hpx/issues/1022
2727 https://github.com/STEllAR-GROUP/hpx/issues/1020
2728 https://github.com/STEllAR-GROUP/hpx/issues/1019
2729 https://github.com/STEllAR-GROUP/hpx/issues/1018
2730 https://github.com/STEllAR-GROUP/hpx/issues/1017
2731 https://github.com/STEllAR-GROUP/hpx/issues/1016
2732 https://github.com/STEllAR-GROUP/hpx/issues/1014
2733 https://github.com/STEllAR-GROUP/hpx/issues/1013
2734 https://github.com/STEllAR-GROUP/hpx/issues/1012
2735 https://github.com/STEllAR-GROUP/hpx/issues/1011
2736 https://github.com/STEllAR-GROUP/hpx/issues/1010
2737 https://github.com/STEllAR-GROUP/hpx/issues/1009
2738 https://github.com/STEllAR-GROUP/hpx/issues/1007
2739 https://github.com/STEllAR-GROUP/hpx/issues/1006
2740 https://github.com/STEllAR-GROUP/hpx/issues/1003
2741 https://github.com/STEllAR-GROUP/hpx/issues/982

- Issue #912[2742] - OS X: component type not found in map
- Issue #663[2743] - Create a buildbot slave based on Clang 3.2/OSX
- Issue #636[2744] - Expose `this_locality::apply<act>(p1, p2);` for local execution
- Issue #197[2745] - Add `--console=address` option for PBS runs
- Issue #175[2746] - Asynchronous AGAS API

2.10.11 *HPX* V0.9.7 (Nov 13, 2013)

We have had over 1000 commits since the last release and we have closed over 180 tickets (bugs, feature requests, etc.).

General changes

- Ported HPX to BlueGene/Q
- Improved HPX support for Xeon/Phi accelerators
- Reimplemented `hpx::bind`, `hpx::tuple`, and `hpx::function` for better performance and better compliance with the C++11 Standard. Added `hpx::mem_fn`.
- Reworked `hpx::when_all` and `hpx::when_any` for better compliance with the ongoing C++ standardization effort, added heterogeneous version for those functions. Added `hpx::when_any_swapped`.
- Added `hpx::copy` as a precursor for a migrate functionality
- Added `hpx::get_ptr` allowing to directly access the memory underlying a given component
- Added the `hpx::lcos::broadcast`, `hpx::lcos::reduce`, and `hpx::lcos::fold` collective operations
- Added `hpx::get_locality_name` allowing to retrieve the name of any of the localities for the application.
- Added support for more flexible thread affinity control from the HPX command line, such as new modes for `--hpx:bind` (`balanced`, `scattered`, `compact`), improved default settings when running multiple localities on the same node.
- Added experimental executors for simpler thread pooling and scheduling. This API may change in the future as it will stay aligned with the ongoing C++ standardization efforts.
- Massively improved the performance of the HPX serialization code. Added partial support for zero copy serialization of array and bitwise-copyable types.
- General performance improvements of the code related to threads and futures.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1005[2747] - Allow to disable array optimizations and zero copy optimizations for each parcelport

[2742] https://github.com/STEllAR-GROUP/hpx/issues/912
[2743] https://github.com/STEllAR-GROUP/hpx/issues/663
[2744] https://github.com/STEllAR-GROUP/hpx/issues/636
[2745] https://github.com/STEllAR-GROUP/hpx/issues/197
[2746] https://github.com/STEllAR-GROUP/hpx/issues/175
[2747] https://github.com/STEllAR-GROUP/hpx/issues/1005

- Issue #1004[2748] - Generate new HPX logo image for the docs
- Issue #1002[2749] - If MPI parcelport is not available, running HPX under mpirun should fail
- Issue #1001[2750] - Zero copy serialization raises assert
- Issue #1000[2751] - Can't connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
- Issue #999[2752] - Optimize `hpx::when_n`
- Issue #998[2753] - Fixed const-correctness
- Issue #997[2754] - Making serialize_buffer::data() type save
- Issue #996[2755] - Memory leak in hpx::lcos::promise
- Issue #995[2756] - Race while registering pre-shutdown functions
- Issue #994[2757] - thread_rescheduling regression test does not compile
- Issue #992[2758] - Correct comments and messages
- Issue #991[2759] - setcap cap_sys_rawio=ep for power profiling causes an HPX application to abort
- Issue #989[2760] - Jacobi hangs during execution
- Issue #988[2761] - multiple_init test is failing
- Issue #986[2762] - Can't call a function called "init" from "main" when using `<hpx/hpx_main.hpp>`
- Issue #984[2763] - Reference counting tests are failing
- Issue #983[2764] - thread_suspension_executor test fails
- Issue #980[2765] - Terminating HPX threads don't leave stack in virgin state
- Issue #979[2766] - Static scheduler not in documents
- Issue #978[2767] - Preprocessing limits are broken
- Issue #977[2768] - Make tests.regressions.lcos.future_hang_on_get shorter
- Issue #976[2769] - Wrong library order in pkgconfig
- Issue #975[2770] - Please reopen #963

2748 https://github.com/STEllAR-GROUP/hpx/issues/1004
2749 https://github.com/STEllAR-GROUP/hpx/issues/1002
2750 https://github.com/STEllAR-GROUP/hpx/issues/1001
2751 https://github.com/STEllAR-GROUP/hpx/issues/1000
2752 https://github.com/STEllAR-GROUP/hpx/issues/999
2753 https://github.com/STEllAR-GROUP/hpx/issues/998
2754 https://github.com/STEllAR-GROUP/hpx/issues/997
2755 https://github.com/STEllAR-GROUP/hpx/issues/996
2756 https://github.com/STEllAR-GROUP/hpx/issues/995
2757 https://github.com/STEllAR-GROUP/hpx/issues/994
2758 https://github.com/STEllAR-GROUP/hpx/issues/992
2759 https://github.com/STEllAR-GROUP/hpx/issues/991
2760 https://github.com/STEllAR-GROUP/hpx/issues/989
2761 https://github.com/STEllAR-GROUP/hpx/issues/988
2762 https://github.com/STEllAR-GROUP/hpx/issues/986
2763 https://github.com/STEllAR-GROUP/hpx/issues/984
2764 https://github.com/STEllAR-GROUP/hpx/issues/983
2765 https://github.com/STEllAR-GROUP/hpx/issues/980
2766 https://github.com/STEllAR-GROUP/hpx/issues/979
2767 https://github.com/STEllAR-GROUP/hpx/issues/978
2768 https://github.com/STEllAR-GROUP/hpx/issues/977
2769 https://github.com/STEllAR-GROUP/hpx/issues/976
2770 https://github.com/STEllAR-GROUP/hpx/issues/975

- Issue #974[2771] - Option pu-offset ignored in fixing_588 branch
- Issue #972[2772] - Cannot use MKL with HPX
- Issue #969[2773] - Non-existent INI files requested on the command line via `--hpx:config` do not cause warnings or errors.
- Issue #968[2774] - Cannot build examples in fixing_588 branch
- Issue #967[2775] - Command line description of `--hpx:queuing` seems wrong
- Issue #966[2776] - `--hpx:print-bind` physical core numbers are wrong
- Issue #965[2777] - Deadlock when building in Release mode
- Issue #963[2778] - Not all worker threads are working
- Issue #962[2779] - Problem with SLURM integration
- Issue #961[2780] - `--hpx:print-bind` outputs incorrect information
- Issue #960[2781] - Fix cut and paste error in documentation of get_thread_priority
- Issue #959[2782] - Change link to boost.atomic in documentation to point to boost.org
- Issue #958[2783] - Undefined reference to intrusive_ptr_release
- Issue #957[2784] - Make tuple standard compliant
- Issue #956[2785] - Segfault with a3382fb
- Issue #955[2786] - `--hpx:nodes` and `--hpx:nodefiles` do not work with foreign nodes
- Issue #954[2787] - Make order of arguments for hpx::async and hpx::broadcast consistent
- Issue #953[2788] - Cannot use MKL with HPX
- Issue #952[2789] - `register_[pre_]shutdown_function` never throw
- Issue #951[2790] - Assert when number of threads is greater than hardware concurrency
- Issue #948[2791] - `HPX_HAVE_GENERIC_CONTEXT_COROUTINES` conflicts with `HPX_HAVE_FIBER_BASED_COROUTINES`
- Issue #947[2792] - Need MPI_THREAD_MULTIPLE for backward compatibility

2771 https://github.com/STEllAR-GROUP/hpx/issues/974
2772 https://github.com/STEllAR-GROUP/hpx/issues/972
2773 https://github.com/STEllAR-GROUP/hpx/issues/969
2774 https://github.com/STEllAR-GROUP/hpx/issues/968
2775 https://github.com/STEllAR-GROUP/hpx/issues/967
2776 https://github.com/STEllAR-GROUP/hpx/issues/966
2777 https://github.com/STEllAR-GROUP/hpx/issues/965
2778 https://github.com/STEllAR-GROUP/hpx/issues/963
2779 https://github.com/STEllAR-GROUP/hpx/issues/962
2780 https://github.com/STEllAR-GROUP/hpx/issues/961
2781 https://github.com/STEllAR-GROUP/hpx/issues/960
2782 https://github.com/STEllAR-GROUP/hpx/issues/959
2783 https://github.com/STEllAR-GROUP/hpx/issues/958
2784 https://github.com/STEllAR-GROUP/hpx/issues/957
2785 https://github.com/STEllAR-GROUP/hpx/issues/956
2786 https://github.com/STEllAR-GROUP/hpx/issues/955
2787 https://github.com/STEllAR-GROUP/hpx/issues/954
2788 https://github.com/STEllAR-GROUP/hpx/issues/953
2789 https://github.com/STEllAR-GROUP/hpx/issues/952
2790 https://github.com/STEllAR-GROUP/hpx/issues/951
2791 https://github.com/STEllAR-GROUP/hpx/issues/948
2792 https://github.com/STEllAR-GROUP/hpx/issues/947

- Issue #946[2793] - HPX does not call `MPI_Finalize`
- Issue #945[2794] - Segfault with `hpx::lcos::broadcast`
- Issue #944[2795] - OS X: assertion `pu_offset_ < hardware_concurrency` failed
- Issue #943[2796] - #include <hpx/hpx_main.hpp> does not work
- Issue #942[2797] - Make the BG/Q work with -O3
- Issue #940[2798] - Use separator when concatenating locality name
- Issue #939[2799] - Refactor MPI parcelport to use `MPI_Wait` instead of multiple `MPI_Test` calls
- Issue #938[2800] - Want to officially access `client_base::gid_`
- Issue #937[2801] - `client_base::gid_` should be private‘‘
- Issue #936[2802] - Want doxygen-like source code index
- Issue #935[2803] - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #933[2804] - Cannot build HPX with Boost 1.54.0
- Issue #932[2805] - Components are destructed too early
- Issue #931[2806] - Make HPX work on BG/Q
- Issue #930[2807] - make git-docs is broken
- Issue #929[2808] - Generating index in docs broken
- Issue #928[2809] - Optimize `hpx::util::static_` for C++11 compilers supporting magic statics
- Issue #924[2810] - Make kill_process_tree (in process.py) more robust on Mac OSX
- Issue #923[2811] - Correct BLAS and RNPL cmake tests
- Issue #922[2812] - Cannot link against BLAS
- Issue #921[2813] - Implement `hpx::mem_fn`
- Issue #920[2814] - Output locality with `--hpx:print-bind`
- Issue #919[2815] - Correct grammar; simplify boolean expressions

[2793] https://github.com/STEllAR-GROUP/hpx/issues/946
[2794] https://github.com/STEllAR-GROUP/hpx/issues/945
[2795] https://github.com/STEllAR-GROUP/hpx/issues/944
[2796] https://github.com/STEllAR-GROUP/hpx/issues/943
[2797] https://github.com/STEllAR-GROUP/hpx/issues/942
[2798] https://github.com/STEllAR-GROUP/hpx/issues/940
[2799] https://github.com/STEllAR-GROUP/hpx/issues/939
[2800] https://github.com/STEllAR-GROUP/hpx/issues/938
[2801] https://github.com/STEllAR-GROUP/hpx/issues/937
[2802] https://github.com/STEllAR-GROUP/hpx/issues/936
[2803] https://github.com/STEllAR-GROUP/hpx/issues/935
[2804] https://github.com/STEllAR-GROUP/hpx/issues/933
[2805] https://github.com/STEllAR-GROUP/hpx/issues/932
[2806] https://github.com/STEllAR-GROUP/hpx/issues/931
[2807] https://github.com/STEllAR-GROUP/hpx/issues/930
[2808] https://github.com/STEllAR-GROUP/hpx/issues/929
[2809] https://github.com/STEllAR-GROUP/hpx/issues/928
[2810] https://github.com/STEllAR-GROUP/hpx/issues/924
[2811] https://github.com/STEllAR-GROUP/hpx/issues/923
[2812] https://github.com/STEllAR-GROUP/hpx/issues/922
[2813] https://github.com/STEllAR-GROUP/hpx/issues/921
[2814] https://github.com/STEllAR-GROUP/hpx/issues/920
[2815] https://github.com/STEllAR-GROUP/hpx/issues/919

- Issue #918[2816] - Link to hello_world.cpp is broken
- Issue #917[2817] - adapt cmake file to new boostbook version
- Issue #916[2818] - fix problem building documentation with xsltproc >= 1.1.27
- Issue #915[2819] - Add another TBBMalloc library search path
- Issue #914[2820] - Build problem with Intel compiler on Stampede (TACC)
- Issue #913[2821] - fix error messages in fibonacci examples
- Issue #911[2822] - Update OS X build instructions
- Issue #910[2823] - Want like to specify MPI_ROOT instead of compiler wrapper script
- Issue #909[2824] - Warning about void* arithmetic
- Issue #908[2825] - Buildbot for MIC is broken
- Issue #906[2826] - Can't use `--hpx:bind=balanced` with multiple MPI processes
- Issue #905[2827] - `--hpx:bind` documentation should describe full grammar
- Issue #904[2828] - Add hpx::lcos::fold and hpx::lcos::inverse_fold collective operation
- Issue #903[2829] - Add `hpx::when_any_swapped()`
- Issue #902[2830] - Add `hpx::lcos::reduce` collective operation
- Issue #901[2831] - Web documentation is not searchable
- Issue #900[2832] - Web documentation for trunk has no index
- Issue #898[2833] - Some tests fail with GCC 4.8.1 and MPI parcel port
- Issue #897[2834] - HWLOC causes failures on Mac
- Issue #896[2835] - pu-offset leads to startup error
- Issue #895[2836] - `hpx::get_locality_name` not defined
- Issue #894[2837] - Race condition at shutdown
- Issue #893[2838] - `--hpx:print-bind` switches std::cout to hexadecimal mode

2816 https://github.com/STEllAR-GROUP/hpx/issues/918
2817 https://github.com/STEllAR-GROUP/hpx/issues/917
2818 https://github.com/STEllAR-GROUP/hpx/issues/916
2819 https://github.com/STEllAR-GROUP/hpx/issues/915
2820 https://github.com/STEllAR-GROUP/hpx/issues/914
2821 https://github.com/STEllAR-GROUP/hpx/issues/913
2822 https://github.com/STEllAR-GROUP/hpx/issues/911
2823 https://github.com/STEllAR-GROUP/hpx/issues/910
2824 https://github.com/STEllAR-GROUP/hpx/issues/909
2825 https://github.com/STEllAR-GROUP/hpx/issues/908
2826 https://github.com/STEllAR-GROUP/hpx/issues/906
2827 https://github.com/STEllAR-GROUP/hpx/issues/905
2828 https://github.com/STEllAR-GROUP/hpx/issues/904
2829 https://github.com/STEllAR-GROUP/hpx/issues/903
2830 https://github.com/STEllAR-GROUP/hpx/issues/902
2831 https://github.com/STEllAR-GROUP/hpx/issues/901
2832 https://github.com/STEllAR-GROUP/hpx/issues/900
2833 https://github.com/STEllAR-GROUP/hpx/issues/898
2834 https://github.com/STEllAR-GROUP/hpx/issues/897
2835 https://github.com/STEllAR-GROUP/hpx/issues/896
2836 https://github.com/STEllAR-GROUP/hpx/issues/895
2837 https://github.com/STEllAR-GROUP/hpx/issues/894
2838 https://github.com/STEllAR-GROUP/hpx/issues/893

- Issue #892[2839] - `hwloc_topology_load` can be expensive – don't call multiple times
- Issue #891[2840] - The documentation for `get_locality_name` is wrong
- Issue #890[2841] - `--hpx:print-bind` should not exit
- Issue #889[2842] - `--hpx:debug-hpx-log=FILE` does not work
- Issue #888[2843] - MPI parcelport does not exit cleanly for –hpx:print-bind
- Issue #887[2844] - Choose thread affinities more cleverly
- Issue #886[2845] - Logging documentation is confusing
- Issue #885[2846] - Two threads are slower than one
- Issue #884[2847] - is_callable failing with member pointers in C++11
- Issue #883[2848] - Need help with is_callable_test
- Issue #882[2849] - tests.regressions.lcos.future_hang_on_get does not terminate
- Issue #881[2850] - tests/regressions/block_matrix/matrix.hh won't compile with GCC 4.8.1
- Issue #880[2851] - HPX does not work on OS X
- Issue #878[2852] - `future::unwrap` triggers assertion
- Issue #877[2853] - "make tests" has build errors on Ubuntu 12.10
- Issue #876[2854] - tcmalloc is used by default, even if it is not present
- Issue #875[2855] - global_fixture is defined in a header file
- Issue #874[2856] - Some tests take very long
- Issue #873[2857] - Add block-matrix code as regression test
- Issue #872[2858] - HPX documentation does not say how to run tests with detailed output
- Issue #871[2859] - All tests fail with "make test"
- Issue #870[2860] - Please explicitly disable serialization in classes that don't support it
- Issue #868[2861] - boost_any test failing

2839 https://github.com/STEllAR-GROUP/hpx/issues/892
2840 https://github.com/STEllAR-GROUP/hpx/issues/891
2841 https://github.com/STEllAR-GROUP/hpx/issues/890
2842 https://github.com/STEllAR-GROUP/hpx/issues/889
2843 https://github.com/STEllAR-GROUP/hpx/issues/888
2844 https://github.com/STEllAR-GROUP/hpx/issues/887
2845 https://github.com/STEllAR-GROUP/hpx/issues/886
2846 https://github.com/STEllAR-GROUP/hpx/issues/885
2847 https://github.com/STEllAR-GROUP/hpx/issues/884
2848 https://github.com/STEllAR-GROUP/hpx/issues/883
2849 https://github.com/STEllAR-GROUP/hpx/issues/882
2850 https://github.com/STEllAR-GROUP/hpx/issues/881
2851 https://github.com/STEllAR-GROUP/hpx/issues/880
2852 https://github.com/STEllAR-GROUP/hpx/issues/878
2853 https://github.com/STEllAR-GROUP/hpx/issues/877
2854 https://github.com/STEllAR-GROUP/hpx/issues/876
2855 https://github.com/STEllAR-GROUP/hpx/issues/875
2856 https://github.com/STEllAR-GROUP/hpx/issues/874
2857 https://github.com/STEllAR-GROUP/hpx/issues/873
2858 https://github.com/STEllAR-GROUP/hpx/issues/872
2859 https://github.com/STEllAR-GROUP/hpx/issues/871
2860 https://github.com/STEllAR-GROUP/hpx/issues/870
2861 https://github.com/STEllAR-GROUP/hpx/issues/868

- Issue #867[2862] - Reduce the number of copies of `hpx::function` arguments
- Issue #863[2863] - Futures should not require a default constructor
- Issue #862[2864] - value_or_error shall not default construct its result
- Issue #861[2865] - `HPX_UNUSED` macro
- Issue #860[2866] - Add functionality to copy construct a component
- Issue #859[2867] - `hpx::endl` should flush
- Issue #858[2868] - Create `hpx::get_ptr<>` allowing to access component implementation
- Issue #855[2869] - Implement `hpx::INVOKE`
- Issue #854[2870] - `hpx/hpx.hpp` does not include `hpx/include/iostreams.hpp`
- Issue #853[2871] - Feature request: null future
- Issue #852[2872] - Feature request: Locality names
- Issue #851[2873] - `hpx::cout` output does not appear on screen
- Issue #849[2874] - All tests fail on OS X after installing
- Issue #848[2875] - Update OS X build instructions
- Issue #846[2876] - Update hpx_external_example
- Issue #845[2877] - Issues with having both debug and release modules in the same directory
- Issue #844[2878] - Create configuration header
- Issue #843[2879] - Tests should use CTest
- Issue #842[2880] - Remove buffer_pool from MPI parcelport
- Issue #841[2881] - Add possibility to broadcast an index with hpx::lcos::broadcast
- Issue #838[2882] - Simplify `util::tuple`
- Issue #837[2883] - Adopt boost::tuple tests for `util::tuple`
- Issue #836[2884] - Adopt boost::function tests for `util::function`

[2862] https://github.com/STEllAR-GROUP/hpx/issues/867
[2863] https://github.com/STEllAR-GROUP/hpx/issues/863
[2864] https://github.com/STEllAR-GROUP/hpx/issues/862
[2865] https://github.com/STEllAR-GROUP/hpx/issues/861
[2866] https://github.com/STEllAR-GROUP/hpx/issues/860
[2867] https://github.com/STEllAR-GROUP/hpx/issues/859
[2868] https://github.com/STEllAR-GROUP/hpx/issues/858
[2869] https://github.com/STEllAR-GROUP/hpx/issues/855
[2870] https://github.com/STEllAR-GROUP/hpx/issues/854
[2871] https://github.com/STEllAR-GROUP/hpx/issues/853
[2872] https://github.com/STEllAR-GROUP/hpx/issues/852
[2873] https://github.com/STEllAR-GROUP/hpx/issues/851
[2874] https://github.com/STEllAR-GROUP/hpx/issues/849
[2875] https://github.com/STEllAR-GROUP/hpx/issues/848
[2876] https://github.com/STEllAR-GROUP/hpx/issues/846
[2877] https://github.com/STEllAR-GROUP/hpx/issues/845
[2878] https://github.com/STEllAR-GROUP/hpx/issues/844
[2879] https://github.com/STEllAR-GROUP/hpx/issues/843
[2880] https://github.com/STEllAR-GROUP/hpx/issues/842
[2881] https://github.com/STEllAR-GROUP/hpx/issues/841
[2882] https://github.com/STEllAR-GROUP/hpx/issues/838
[2883] https://github.com/STEllAR-GROUP/hpx/issues/837
[2884] https://github.com/STEllAR-GROUP/hpx/issues/836

- Issue #835[2885] - Tuple interface missing pieces
- Issue #833[2886] - Partially preprocessing files not working
- Issue #832[2887] - Native papi counters do not work with wild cards
- Issue #831[2888] - Arithmetics counter fails if only one parameter is given
- Issue #830[2889] - Convert hpx::util::function to use new scheme for serializing its base pointer
- Issue #829[2890] - Consistently use `decay<T>` instead of `remove_const< remove_reference<T>>`
- Issue #828[2891] - Update future implementation to N3721 and N3722
- Issue #827[2892] - Enable MPI parcelport for bootstrapping whenever application was started using mpirun
- Issue #826[2893] - Support command line option `--hpx:print-bind` even if `--hpx::bind` was not used
- Issue #825[2894] - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- Issue #824[2895] - Enable lambda functions to be used with hpx::async/hpx::apply
- Issue #823[2896] - Using a hashing filter
- Issue #822[2897] - Silence unused variable warning
- Issue #821[2898] - Detect if a function object is callable with given arguments
- Issue #820[2899] - Allow wildcards to be used for performance counter names
- Issue #819[2900] - Make the AGAS symbolic name registry distributed
- Issue #818[2901] - Add future::then() overload taking an executor
- Issue #817[2902] - Fixed typo
- Issue #815[2903] - Create an lco that is performing an efficient broadcast of actions
- Issue #814[2904] - Papi counters cannot specify thread#* to get the counts for all threads
- Issue #813[2905] - Scoped unlock
- Issue #811[2906] - simple_central_tuplespace_client run error
- Issue #810[2907] - ostream error when << any objects

[2885] https://github.com/STEllAR-GROUP/hpx/issues/835
[2886] https://github.com/STEllAR-GROUP/hpx/issues/833
[2887] https://github.com/STEllAR-GROUP/hpx/issues/832
[2888] https://github.com/STEllAR-GROUP/hpx/issues/831
[2889] https://github.com/STEllAR-GROUP/hpx/issues/830
[2890] https://github.com/STEllAR-GROUP/hpx/issues/829
[2891] https://github.com/STEllAR-GROUP/hpx/issues/828
[2892] https://github.com/STEllAR-GROUP/hpx/issues/827
[2893] https://github.com/STEllAR-GROUP/hpx/issues/826
[2894] https://github.com/STEllAR-GROUP/hpx/issues/825
[2895] https://github.com/STEllAR-GROUP/hpx/issues/824
[2896] https://github.com/STEllAR-GROUP/hpx/issues/823
[2897] https://github.com/STEllAR-GROUP/hpx/issues/822
[2898] https://github.com/STEllAR-GROUP/hpx/issues/821
[2899] https://github.com/STEllAR-GROUP/hpx/issues/820
[2900] https://github.com/STEllAR-GROUP/hpx/issues/819
[2901] https://github.com/STEllAR-GROUP/hpx/issues/818
[2902] https://github.com/STEllAR-GROUP/hpx/issues/817
[2903] https://github.com/STEllAR-GROUP/hpx/issues/815
[2904] https://github.com/STEllAR-GROUP/hpx/issues/814
[2905] https://github.com/STEllAR-GROUP/hpx/issues/813
[2906] https://github.com/STEllAR-GROUP/hpx/issues/811
[2907] https://github.com/STEllAR-GROUP/hpx/issues/810

- Issue #809[2908] - Optimize parcel serialization
- Issue #808[2909] - HPX applications throw exception when executed from the build directory
- Issue #807[2910] - Create performance counters exposing overall AGAS statistics
- Issue #795[2911] - Create timed make_ready_future
- Issue #794[2912] - Create heterogeneous `when_all`/`when_any`/etc.
- Issue #721[2913] - Make HPX usable for Xeon Phi
- Issue #694[2914] - CMake should complain if you attempt to build an example without its dependencies
- Issue #692[2915] - SLURM support broken
- Issue #683[2916] - python/hpx/process.py imports epoll on all platforms
- Issue #619[2917] - Automate the doc building process
- Issue #600[2918] - GTC performance broken
- Issue #577[2919] - Allow for zero copy serialization/networking
- Issue #551[2920] - Change executable names to have debug postfix in Debug builds
- Issue #544[2921] - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init
- Issue #534[2922] - `hpx::init` should take functions by `std::function` and should accept all forms of hpx_main
- Issue #508[2923] - FindPackage fails to set FOO_LIBRARY_DIR
- Issue #506[2924] - Add cmake support to generate ini files for external applications
- Issue #470[2925] - Changing build-type after configure does not update boost library names
- Issue #453[2926] - Document `hpx_run_tests.py`
- Issue #445[2927] - Significant performance mismatch between MPI and HPX in SMP for allgather example
- Issue #443[2928] - Make docs viewable from build directory
- Issue #421[2929] - Support multiple HPX instances per node in a batch environment like PBS or SLURM
- Issue #316[2930] - Add message size limitation

[2908] https://github.com/STEllAR-GROUP/hpx/issues/809
[2909] https://github.com/STEllAR-GROUP/hpx/issues/808
[2910] https://github.com/STEllAR-GROUP/hpx/issues/807
[2911] https://github.com/STEllAR-GROUP/hpx/issues/795
[2912] https://github.com/STEllAR-GROUP/hpx/issues/794
[2913] https://github.com/STEllAR-GROUP/hpx/issues/721
[2914] https://github.com/STEllAR-GROUP/hpx/issues/694
[2915] https://github.com/STEllAR-GROUP/hpx/issues/692
[2916] https://github.com/STEllAR-GROUP/hpx/issues/683
[2917] https://github.com/STEllAR-GROUP/hpx/issues/619
[2918] https://github.com/STEllAR-GROUP/hpx/issues/600
[2919] https://github.com/STEllAR-GROUP/hpx/issues/577
[2920] https://github.com/STEllAR-GROUP/hpx/issues/551
[2921] https://github.com/STEllAR-GROUP/hpx/issues/544
[2922] https://github.com/STEllAR-GROUP/hpx/issues/534
[2923] https://github.com/STEllAR-GROUP/hpx/issues/508
[2924] https://github.com/STEllAR-GROUP/hpx/issues/506
[2925] https://github.com/STEllAR-GROUP/hpx/issues/470
[2926] https://github.com/STEllAR-GROUP/hpx/issues/453
[2927] https://github.com/STEllAR-GROUP/hpx/issues/445
[2928] https://github.com/STEllAR-GROUP/hpx/issues/443
[2929] https://github.com/STEllAR-GROUP/hpx/issues/421
[2930] https://github.com/STEllAR-GROUP/hpx/issues/316

- Issue #249[2931] - Clean up locking code in big boot barrier
- Issue #136[2932] - Persistent CMake variables need to be marked as cache variables

2.10.12 *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 fetures 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[2933] and related proposals to the C++ standardization committee (such as N3632[2934] and N3857[2935]).
- We implemented a first version of a distributed AGAS service which essentially eliminates all explicit AGAS network traffic.
- We created a native ibverbs parcelport allowing to take advantage of the superior latency and bandwidth characteristics of Infiniband networks.
- We successfully ported *HPX* to the Xeon Phi platform.
- Support for the SLURM scheduling system was implemented.
- Major efforts have been dedicated to improving the performance counter framework, numerous new counters were implemented and new APIs were added.
- We added a modular parcel compression system allowing to improve bandwidth utilization (by reducing the overall size of the tranferred 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[2936]. 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[2937]).

[2931] https://github.com/STEllAR-GROUP/hpx/issues/249
[2932] https://github.com/STEllAR-GROUP/hpx/issues/136
[2933] http://www.open-std.org/jtc1/sc22/wg21
[2934] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3632.html
[2935] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf
[2936] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf
[2937] https://en.wikipedia.org/wiki/Linda_(coordination_language)

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[2938] - make (all) in examples folder does nothing
- Issue #805[2939] - Adding the introduction and fixing DOCBOOK dependencies for Windows use
- Issue #804[2940] - Add stackless (non-suspendable) thread type
- Issue #803[2941] - Create proper serialization support functions for util::tuple
- Issue #800[2942] - Add possibility to disable array optimizations during serialization
- Issue #798[2943] - HPX_LIMIT does not work for local dataflow
- Issue #797[2944] - Create a parcelport which uses MPI
- Issue #796[2945] - Problem with Large Numbers of Threads
- Issue #793[2946] - Changing dataflow test case to hang consistently
- Issue #792[2947] - CMake Error
- Issue #791[2948] - Problems with local::dataflow
- Issue #790[2949] - wait_for() doesn't compile
- Issue #789[2950] - HPX with Intel compiler segfaults
- Issue #788[2951] - Intel compiler support
- Issue #787[2952] - Fixed SFINAEd specializations
- Issue #786[2953] - Memory issues during benchmarking.
- Issue #785[2954] - Create an API allowing to register external threads with HPX
- Issue #784[2955] - util::plugin is throwing an error when a symbol is not found
- Issue #783[2956] - How does hpx:bind work?
- Issue #782[2957] - Added quotes around STRING REPLACE potentially empty arguments
- Issue #781[2958] - Make sure no exceptions propagate into the thread manager

[2938] https://github.com/STEllAR-GROUP/hpx/issues/806
[2939] https://github.com/STEllAR-GROUP/hpx/issues/805
[2940] https://github.com/STEllAR-GROUP/hpx/issues/804
[2941] https://github.com/STEllAR-GROUP/hpx/issues/803
[2942] https://github.com/STEllAR-GROUP/hpx/issues/800
[2943] https://github.com/STEllAR-GROUP/hpx/issues/798
[2944] https://github.com/STEllAR-GROUP/hpx/issues/797
[2945] https://github.com/STEllAR-GROUP/hpx/issues/796
[2946] https://github.com/STEllAR-GROUP/hpx/issues/793
[2947] https://github.com/STEllAR-GROUP/hpx/issues/792
[2948] https://github.com/STEllAR-GROUP/hpx/issues/791
[2949] https://github.com/STEllAR-GROUP/hpx/issues/790
[2950] https://github.com/STEllAR-GROUP/hpx/issues/789
[2951] https://github.com/STEllAR-GROUP/hpx/issues/788
[2952] https://github.com/STEllAR-GROUP/hpx/issues/787
[2953] https://github.com/STEllAR-GROUP/hpx/issues/786
[2954] https://github.com/STEllAR-GROUP/hpx/issues/785
[2955] https://github.com/STEllAR-GROUP/hpx/issues/784
[2956] https://github.com/STEllAR-GROUP/hpx/issues/783
[2957] https://github.com/STEllAR-GROUP/hpx/issues/782
[2958] https://github.com/STEllAR-GROUP/hpx/issues/781

- Issue #780[2959] - Allow arithmetics performance counters to expand its parameters
- Issue #779[2960] - Test case for 778
- Issue #778[2961] - Swapping futures segfaults
- Issue #777[2962] - hpx::lcos::details::when_xxx don't restore completion handlers
- Issue #776[2963] - Compiler chokes on dataflow overload with launch policy
- Issue #775[2964] - Runtime error with local dataflow (copying futures?)
- Issue #774[2965] - Using local dataflow without explicit namespace
- Issue #773[2966] - Local dataflow with unwrap: functor operators need to be const
- Issue #772[2967] - Allow (remote) actions to return a future
- Issue #771[2968] - Setting HPX_LIMIT gives huge boost MPL errors
- Issue #770[2969] - Add launch policy to (local) dataflow
- Issue #769[2970] - Make compile time configuration information available
- Issue #768[2971] - Const correctness problem in local dataflow
- Issue #767[2972] - Add launch policies to async
- Issue #766[2973] - Mark data structures for optimized (array based) serialization
- Issue #765[2974] - Align hpx::any with N3508: Any Library Proposal (Revision 2)
- Issue #764[2975] - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations
- Issue #762[2976] - added a human readable output for the ping pong example
- Issue #761[2977] - Ambiguous typename when constructing derived component
- Issue #760[2978] - Simple components can not be derived
- Issue #759[2979] - make install doesn't give a complete install
- Issue #758[2980] - Stack overflow when using locking_hook<>
- Issue #757[2981] - copy paste error; unsupported function overloading

[2959] https://github.com/STEllAR-GROUP/hpx/issues/780
[2960] https://github.com/STEllAR-GROUP/hpx/issues/779
[2961] https://github.com/STEllAR-GROUP/hpx/issues/778
[2962] https://github.com/STEllAR-GROUP/hpx/issues/777
[2963] https://github.com/STEllAR-GROUP/hpx/issues/776
[2964] https://github.com/STEllAR-GROUP/hpx/issues/775
[2965] https://github.com/STEllAR-GROUP/hpx/issues/774
[2966] https://github.com/STEllAR-GROUP/hpx/issues/773
[2967] https://github.com/STEllAR-GROUP/hpx/issues/772
[2968] https://github.com/STEllAR-GROUP/hpx/issues/771
[2969] https://github.com/STEllAR-GROUP/hpx/issues/770
[2970] https://github.com/STEllAR-GROUP/hpx/issues/769
[2971] https://github.com/STEllAR-GROUP/hpx/issues/768
[2972] https://github.com/STEllAR-GROUP/hpx/issues/767
[2973] https://github.com/STEllAR-GROUP/hpx/issues/766
[2974] https://github.com/STEllAR-GROUP/hpx/issues/765
[2975] https://github.com/STEllAR-GROUP/hpx/issues/764
[2976] https://github.com/STEllAR-GROUP/hpx/issues/762
[2977] https://github.com/STEllAR-GROUP/hpx/issues/761
[2978] https://github.com/STEllAR-GROUP/hpx/issues/760
[2979] https://github.com/STEllAR-GROUP/hpx/issues/759
[2980] https://github.com/STEllAR-GROUP/hpx/issues/758
[2981] https://github.com/STEllAR-GROUP/hpx/issues/757

- Issue #756[2982] - GTCX runtime issue in Gordon
- Issue #755[2983] - Papi counters don't work with reset and evaluate API's
- Issue #753[2984] - cmake bugfix and improved component action docs
- Issue #752[2985] - hpx simple component docs
- Issue #750[2986] - Add hpx::util::any
- Issue #749[2987] - Thread phase counter is not reset
- Issue #748[2988] - Memory performance counter are not registered
- Issue #747[2989] - Create performance counters exposing arithmetic operations
- Issue #745[2990] - apply_callback needs to invoke callback when applied locally
- Issue #744[2991] - CMake fixes
- Issue #743[2992] - Problem Building github version of HPX
- Issue #742[2993] - Remove HPX_STD_BIND
- Issue #741[2994] - assertion 'px != 0' failed: HPX(assertion_failure) for low numbers of OS threads
- Issue #739[2995] - Performance counters do not count to the end of the program or evalution
- Issue #738[2996] - Dedicated AGAS server runs don't work; console ignores -a option.
- Issue #737[2997] - Missing bind overloads
- Issue #736[2998] - Performance counter wildcards do not always work
- Issue #735[2999] - Create native ibverbs parcelport based on rdma operations
- Issue #734[3000] - Threads stolen performance counter total is incorrect
- Issue #733[3001] - Test benchmarks need to be checked and fixed
- Issue #732[3002] - Build fails with Mac, using mac ports clang-3.3 on latest git branch
- Issue #731[3003] - Add global start/stop API for performance counters
- Issue #730[3004] - Performance counter values are apparently incorrect

[2982] https://github.com/STEllAR-GROUP/hpx/issues/756
[2983] https://github.com/STEllAR-GROUP/hpx/issues/755
[2984] https://github.com/STEllAR-GROUP/hpx/issues/753
[2985] https://github.com/STEllAR-GROUP/hpx/issues/752
[2986] https://github.com/STEllAR-GROUP/hpx/issues/750
[2987] https://github.com/STEllAR-GROUP/hpx/issues/749
[2988] https://github.com/STEllAR-GROUP/hpx/issues/748
[2989] https://github.com/STEllAR-GROUP/hpx/issues/747
[2990] https://github.com/STEllAR-GROUP/hpx/issues/745
[2991] https://github.com/STEllAR-GROUP/hpx/issues/744
[2992] https://github.com/STEllAR-GROUP/hpx/issues/743
[2993] https://github.com/STEllAR-GROUP/hpx/issues/742
[2994] https://github.com/STEllAR-GROUP/hpx/issues/741
[2995] https://github.com/STEllAR-GROUP/hpx/issues/739
[2996] https://github.com/STEllAR-GROUP/hpx/issues/738
[2997] https://github.com/STEllAR-GROUP/hpx/issues/737
[2998] https://github.com/STEllAR-GROUP/hpx/issues/736
[2999] https://github.com/STEllAR-GROUP/hpx/issues/735
[3000] https://github.com/STEllAR-GROUP/hpx/issues/734
[3001] https://github.com/STEllAR-GROUP/hpx/issues/733
[3002] https://github.com/STEllAR-GROUP/hpx/issues/732
[3003] https://github.com/STEllAR-GROUP/hpx/issues/731
[3004] https://github.com/STEllAR-GROUP/hpx/issues/730

- Issue #729[3005] - Unhandled switch
- Issue #728[3006] - Serialization of hpx::util::function between two localities causes seg faults
- Issue #727[3007] - Memory counters on Mac OS X
- Issue #725[3008] - Restore original thread priority on resume
- Issue #724[3009] - Performance benchmarks do not depend on main HPX libraries
- Issue #723[3010] - [teletype]–hpx:nodes=‘‘cat $PBS_NODEFILE‘‘ works; –hpx:nodefile=$PBS_NODEFILE does not.[c++]
- Issue #722[3011] - Fix binding const member functions as actions
- Issue #719[3012] - Create performance counter exposing compression ratio
- Issue #718[3013] - Add possibility to compress parcel data
- Issue #717[3014] - strip_credit_from_gid has misleading semantics
- Issue #716[3015] - Non-option arguments to programs run using `pbsdsh` must be before `--hpx:nodes`, contrary to directions
- Issue #715[3016] - Re-thrown exceptions should retain the original call site
- Issue #714[3017] - failed assertion in debug mode
- Issue #713[3018] - Add performance counters monitoring connection caches
- Issue #712[3019] - Adjust parcel related performance counters to be connection type specific
- Issue #711[3020] - configuration failure
- Issue #710[3021] - Error "timed out while trying to find room in the connection cache" when trying to start multiple localities on a single computer
- Issue #709[3022] - Add new thread state 'staged' referring to task descriptions
- Issue #708[3023] - Detect/mitigate bad non-system installs of GCC on Redhat systems
- Issue #707[3024] - Many examples do not link with Git HEAD version
- Issue #706[3025] - `hpx::init` removes portions of non-option command line arguments before last = sign
- Issue #705[3026] - Create rolling average and median aggregating performance counters

3005 https://github.com/STEllAR-GROUP/hpx/issues/729
3006 https://github.com/STEllAR-GROUP/hpx/issues/728
3007 https://github.com/STEllAR-GROUP/hpx/issues/727
3008 https://github.com/STEllAR-GROUP/hpx/issues/725
3009 https://github.com/STEllAR-GROUP/hpx/issues/724
3010 https://github.com/STEllAR-GROUP/hpx/issues/723
3011 https://github.com/STEllAR-GROUP/hpx/issues/722
3012 https://github.com/STEllAR-GROUP/hpx/issues/719
3013 https://github.com/STEllAR-GROUP/hpx/issues/718
3014 https://github.com/STEllAR-GROUP/hpx/issues/717
3015 https://github.com/STEllAR-GROUP/hpx/issues/716
3016 https://github.com/STEllAR-GROUP/hpx/issues/715
3017 https://github.com/STEllAR-GROUP/hpx/issues/714
3018 https://github.com/STEllAR-GROUP/hpx/issues/713
3019 https://github.com/STEllAR-GROUP/hpx/issues/712
3020 https://github.com/STEllAR-GROUP/hpx/issues/711
3021 https://github.com/STEllAR-GROUP/hpx/issues/710
3022 https://github.com/STEllAR-GROUP/hpx/issues/709
3023 https://github.com/STEllAR-GROUP/hpx/issues/708
3024 https://github.com/STEllAR-GROUP/hpx/issues/707
3025 https://github.com/STEllAR-GROUP/hpx/issues/706
3026 https://github.com/STEllAR-GROUP/hpx/issues/705

- Issue #704[3027] - Create performance counter to expose thread queue waiting time
- Issue #703[3028] - Add support to HPX build system to find librcrtool.a and related headers
- Issue #699[3029] - Generalize instrumentation support
- Issue #698[3030] - compilation failure with hwloc absent
- Issue #697[3031] - Performance counter counts should be zero indexed
- Issue #696[3032] - Distributed problem
- Issue #695[3033] - Bad perf counter time printed
- Issue #693[3034] - `--help` doesn't print component specific command line options
- Issue #692[3035] - SLURM support broken
- Issue #691[3036] - exception while executing any application linked with hwloc
- Issue #690[3037] - thread_id_test and thread_launcher_test failing
- Issue #689[3038] - Make the buildbots use hwloc
- Issue #687[3039] - compilation error fix (hwloc_topology)
- Issue #686[3040] - Linker Error for Applications
- Issue #684[3041] - Pinning of service thread fails when number of worker threads equals the number of cores
- Issue #682[3042] - Add performance counters exposing number of stolen threads
- Issue #681[3043] - Add apply_continue for asynchronous chaining of actions
- Issue #679[3044] - Remove obsolete async_callback API functions
- Issue #678[3045] - Add new API for setting/triggering LCOs
- Issue #677[3046] - Add async_continue for true continuation style actions
- Issue #676[3047] - Buildbot for gcc 4.4 broken
- Issue #675[3048] - Partial preprocessing broken
- Issue #674[3049] - HPX segfaults when built with gcc 4.7

[3027] https://github.com/STEllAR-GROUP/hpx/issues/704
[3028] https://github.com/STEllAR-GROUP/hpx/issues/703
[3029] https://github.com/STEllAR-GROUP/hpx/issues/699
[3030] https://github.com/STEllAR-GROUP/hpx/issues/698
[3031] https://github.com/STEllAR-GROUP/hpx/issues/697
[3032] https://github.com/STEllAR-GROUP/hpx/issues/696
[3033] https://github.com/STEllAR-GROUP/hpx/issues/695
[3034] https://github.com/STEllAR-GROUP/hpx/issues/693
[3035] https://github.com/STEllAR-GROUP/hpx/issues/692
[3036] https://github.com/STEllAR-GROUP/hpx/issues/691
[3037] https://github.com/STEllAR-GROUP/hpx/issues/690
[3038] https://github.com/STEllAR-GROUP/hpx/issues/689
[3039] https://github.com/STEllAR-GROUP/hpx/issues/687
[3040] https://github.com/STEllAR-GROUP/hpx/issues/686
[3041] https://github.com/STEllAR-GROUP/hpx/issues/684
[3042] https://github.com/STEllAR-GROUP/hpx/issues/682
[3043] https://github.com/STEllAR-GROUP/hpx/issues/681
[3044] https://github.com/STEllAR-GROUP/hpx/issues/679
[3045] https://github.com/STEllAR-GROUP/hpx/issues/678
[3046] https://github.com/STEllAR-GROUP/hpx/issues/677
[3047] https://github.com/STEllAR-GROUP/hpx/issues/676
[3048] https://github.com/STEllAR-GROUP/hpx/issues/675
[3049] https://github.com/STEllAR-GROUP/hpx/issues/674

- Issue #673[3050] - `use_guard_pages` has inconsistent preprocessor guards
- Issue #672[3051] - External build breaks if library path has spaces
- Issue #671[3052] - release tarballs are tarbombs
- Issue #670[3053] - CMake won't find Boost headers in layout=versioned install
- Issue #669[3054] - Links in docs to source files broken if not installed
- Issue #667[3055] - Not reading ini file properly
- Issue #664[3056] - Adapt new meanings of 'const' and 'mutable'
- Issue #661[3057] - Implement BTL Parcel port
- Issue #655[3058] - Make HPX work with the "decltype" result_of
- Issue #647[3059] - documentation for specifying the number of high priority threads `--hpx:high-priority-threads`
- Issue #643[3060] - Error parsing host file
- Issue #642[3061] - HWLoc issue with TAU
- Issue #639[3062] - Logging potentially suspends a running thread
- Issue #634[3063] - Improve error reporting from parcel layer
- Issue #627[3064] - Add tests for async and apply overloads that accept regular C++ functions
- Issue #626[3065] - hpx/future.hpp header
- Issue #601[3066] - Intel support
- Issue #557[3067] - Remove action codes
- Issue #531[3068] - AGAS request and response classes should use switch statements
- Issue #529[3069] - Investigate the state of hwloc support
- Issue #526[3070] - Make HPX aware of hyper-threading
- Issue #518[3071] - Create facilities allowing to use plain arrays as action arguments
- Issue #473[3072] - hwloc thread binding is broken on CPUs with hyperthreading

[3050] https://github.com/STEllAR-GROUP/hpx/issues/673
[3051] https://github.com/STEllAR-GROUP/hpx/issues/672
[3052] https://github.com/STEllAR-GROUP/hpx/issues/671
[3053] https://github.com/STEllAR-GROUP/hpx/issues/670
[3054] https://github.com/STEllAR-GROUP/hpx/issues/669
[3055] https://github.com/STEllAR-GROUP/hpx/issues/667
[3056] https://github.com/STEllAR-GROUP/hpx/issues/664
[3057] https://github.com/STEllAR-GROUP/hpx/issues/661
[3058] https://github.com/STEllAR-GROUP/hpx/issues/655
[3059] https://github.com/STEllAR-GROUP/hpx/issues/647
[3060] https://github.com/STEllAR-GROUP/hpx/issues/643
[3061] https://github.com/STEllAR-GROUP/hpx/issues/642
[3062] https://github.com/STEllAR-GROUP/hpx/issues/639
[3063] https://github.com/STEllAR-GROUP/hpx/issues/634
[3064] https://github.com/STEllAR-GROUP/hpx/issues/627
[3065] https://github.com/STEllAR-GROUP/hpx/issues/626
[3066] https://github.com/STEllAR-GROUP/hpx/issues/601
[3067] https://github.com/STEllAR-GROUP/hpx/issues/557
[3068] https://github.com/STEllAR-GROUP/hpx/issues/531
[3069] https://github.com/STEllAR-GROUP/hpx/issues/529
[3070] https://github.com/STEllAR-GROUP/hpx/issues/526
[3071] https://github.com/STEllAR-GROUP/hpx/issues/518
[3072] https://github.com/STEllAR-GROUP/hpx/issues/473

- Issue #383[3073] - Change result type detection for hpx::util::bind to use result_of protocol
- Issue #341[3074] - Consolidate route code
- Issue #219[3075] - Only copy arguments into actions once
- Issue #177[3076] - Implement distributed AGAS
- Issue #43[3077] - Support for Darwin (Xcode + Clang)

2.10.13 *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 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[3078] 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[3079]).

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.

[3073] https://github.com/STEllAR-GROUP/hpx/issues/383
[3074] https://github.com/STEllAR-GROUP/hpx/issues/341
[3075] https://github.com/STEllAR-GROUP/hpx/issues/219
[3076] https://github.com/STEllAR-GROUP/hpx/issues/177
[3077] https://github.com/STEllAR-GROUP/hpx/issues/43
[3078] https://github.com/STEllAR-GROUP/hpx/
[3079] https://www.boost.org/LICENSE_1_0.txt

- Issue #666[3080] - Segfault on calling hpx::finalize twice
- Issue #665[3081] - Adding declaration num_of_cores
- Issue #662[3082] - pkgconfig is building wrong
- Issue #660[3083] - Need uninterrupt function
- Issue #659[3084] - Move our logging library into a different namespace
- Issue #658[3085] - Dynamic performance counter types are broken
- Issue #657[3086] - HPX v0.9.5 (RC1) hello_world example segfaulting
- Issue #656[3087] - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- Issue #654[3088] - Integrate the Boost auto_index tool with documentation
- Issue #653[3089] - Make HPX build on OS X + Clang + libc++
- Issue #651[3090] - Add fine-grained control for thread pinning
- Issue #650[3091] - Command line no error message when using -hpx:(anything)
- Issue #645[3092] - Command line aliases don't work in [teletype]``@file``[c++]
- Issue #644[3093] - Terminated threads are not always properly cleaned up
- Issue #640[3094] - `future_data<T>::set_on_completed_` used without locks
- Issue #638[3095] - hpx build with intel compilers fails on linux
- Issue #637[3096] - –copy-dt-needed-entries breaks with gold
- Issue #635[3097] - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- Issue #633[3098] - Re-add examples to final 0.9.5 release
- Issue #632[3099] - Example `thread_aware_timer` is broken
- Issue #631[3100] - FFT application throws error in parcellayer
- Issue #630[3101] - Event synchronization example is broken
- Issue #629[3102] - Waiting on futures hangs

[3080] https://github.com/STEllAR-GROUP/hpx/issues/666
[3081] https://github.com/STEllAR-GROUP/hpx/issues/665
[3082] https://github.com/STEllAR-GROUP/hpx/issues/662
[3083] https://github.com/STEllAR-GROUP/hpx/issues/660
[3084] https://github.com/STEllAR-GROUP/hpx/issues/659
[3085] https://github.com/STEllAR-GROUP/hpx/issues/658
[3086] https://github.com/STEllAR-GROUP/hpx/issues/657
[3087] https://github.com/STEllAR-GROUP/hpx/issues/656
[3088] https://github.com/STEllAR-GROUP/hpx/issues/654
[3089] https://github.com/STEllAR-GROUP/hpx/issues/653
[3090] https://github.com/STEllAR-GROUP/hpx/issues/651
[3091] https://github.com/STEllAR-GROUP/hpx/issues/650
[3092] https://github.com/STEllAR-GROUP/hpx/issues/645
[3093] https://github.com/STEllAR-GROUP/hpx/issues/644
[3094] https://github.com/STEllAR-GROUP/hpx/issues/640
[3095] https://github.com/STEllAR-GROUP/hpx/issues/638
[3096] https://github.com/STEllAR-GROUP/hpx/issues/637
[3097] https://github.com/STEllAR-GROUP/hpx/issues/635
[3098] https://github.com/STEllAR-GROUP/hpx/issues/633
[3099] https://github.com/STEllAR-GROUP/hpx/issues/632
[3100] https://github.com/STEllAR-GROUP/hpx/issues/631
[3101] https://github.com/STEllAR-GROUP/hpx/issues/630
[3102] https://github.com/STEllAR-GROUP/hpx/issues/629

- Issue #628[3103] - Add an `HPX_ALWAYS_ASSERT` macro
- Issue #625[3104] - Port coroutines context switch benchmark
- Issue #621[3105] - New INI section for stack sizes
- Issue #618[3106] - pkg_config support does not work with a HPX debug build
- Issue #617[3107] - hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: 'get_thread_id' was not declared in this scope
- Issue #616[3108] - Change wait_xxx not to use locking
- Issue #615[3109] - Revert visibility 'fix' (fb0b6b8245dad1127b0c25ebafd9386b3945cca9)
- Issue #614[3110] - Fix Dataflow linker error
- Issue #613[3111] - find_here should throw an exception on failure
- Issue #612[3112] - Thread phase doesn't show up in debug mode
- Issue #611[3113] - Make stack guard pages configurable at runtime (initialization time)
- Issue #610[3114] - Co-Locate Components
- Issue #609[3115] - future_overhead
- Issue #608[3116] - `--hpx:list-counter-infos` problem
- Issue #607[3117] - Update Boost.Context based backend for coroutines
- Issue #606[3118] - 1d_wave_equation is not working
- Issue #605[3119] - Any C++ function that has serializable arguments and a serializable return type should be remotable
- Issue #604[3120] - Connecting localities isn't working anymore
- Issue #603[3121] - Do not verify any ini entries read from a file
- Issue #602[3122] - Rename argument_size to type_size/ added implementation to get parcel size
- Issue #599[3123] - Enable locality specific command line options
- Issue #598[3124] - Need an API that accesses the performance counter reporting the system uptime

[3103] https://github.com/STEllAR-GROUP/hpx/issues/628
[3104] https://github.com/STEllAR-GROUP/hpx/issues/625
[3105] https://github.com/STEllAR-GROUP/hpx/issues/621
[3106] https://github.com/STEllAR-GROUP/hpx/issues/618
[3107] https://github.com/STEllAR-GROUP/hpx/issues/617
[3108] https://github.com/STEllAR-GROUP/hpx/issues/616
[3109] https://github.com/STEllAR-GROUP/hpx/issues/615
[3110] https://github.com/STEllAR-GROUP/hpx/issues/614
[3111] https://github.com/STEllAR-GROUP/hpx/issues/613
[3112] https://github.com/STEllAR-GROUP/hpx/issues/612
[3113] https://github.com/STEllAR-GROUP/hpx/issues/611
[3114] https://github.com/STEllAR-GROUP/hpx/issues/610
[3115] https://github.com/STEllAR-GROUP/hpx/issues/609
[3116] https://github.com/STEllAR-GROUP/hpx/issues/608
[3117] https://github.com/STEllAR-GROUP/hpx/issues/607
[3118] https://github.com/STEllAR-GROUP/hpx/issues/606
[3119] https://github.com/STEllAR-GROUP/hpx/issues/605
[3120] https://github.com/STEllAR-GROUP/hpx/issues/604
[3121] https://github.com/STEllAR-GROUP/hpx/issues/603
[3122] https://github.com/STEllAR-GROUP/hpx/issues/602
[3123] https://github.com/STEllAR-GROUP/hpx/issues/599
[3124] https://github.com/STEllAR-GROUP/hpx/issues/598

- Issue #597[3125] - compiling on ranger
- Issue #595[3126] - I need a place to store data in a thread self pointer
- Issue #594[3127] - 32/64 interoperability
- Issue #593[3128] - Warn if logging is disabled at compile time but requested at runtime
- Issue #592[3129] - Add optional argument value to `--hpx:list-counters` and `--hpx:list-counter-infos`
- Issue #591[3130] - Allow for wildcards in performance counter names specified with `--hpx:print-counter`
- Issue #590[3131] - Local promise semantic differences
- Issue #589[3132] - Create API to query performance counter names
- Issue #587[3133] - Add get_num_localities and get_num_threads to AGAS API
- Issue #586[3134] - Adjust local AGAS cache size based on number of localities
- Issue #585[3135] - Error while using counters in HPX
- Issue #584[3136] - counting argument size of actions, initial pass.
- Issue #581[3137] - Remove `RemoteResult` template parameter for `future<>`
- Issue #580[3138] - Add possibility to hook into actions
- Issue #578[3139] - Use angle brackets in HPX error dumps
- Issue #576[3140] - Exception incorrectly thrown when `--help` is used
- Issue #575[3141] - HPX(bad_component_type) with gcc 4.7.2 and boost 1.51
- Issue #574[3142] - `--hpx:connect` command line parameter not working correctly
- Issue #571[3143] - `hpx::wait()` (callback version) should pass the future to the callback function
- Issue #570[3144] - `hpx::wait` should operate on `boost::arrays` and `std::lists`
- Issue #569[3145] - Add a logging sink for Android
- Issue #568[3146] - 2-argument version of `HPX_DEFINE_COMPONENT_ACTION`
- Issue #567[3147] - Connecting to a running HPX application works only once

[3125] https://github.com/STEllAR-GROUP/hpx/issues/597
[3126] https://github.com/STEllAR-GROUP/hpx/issues/595
[3127] https://github.com/STEllAR-GROUP/hpx/issues/594
[3128] https://github.com/STEllAR-GROUP/hpx/issues/593
[3129] https://github.com/STEllAR-GROUP/hpx/issues/592
[3130] https://github.com/STEllAR-GROUP/hpx/issues/591
[3131] https://github.com/STEllAR-GROUP/hpx/issues/590
[3132] https://github.com/STEllAR-GROUP/hpx/issues/589
[3133] https://github.com/STEllAR-GROUP/hpx/issues/587
[3134] https://github.com/STEllAR-GROUP/hpx/issues/586
[3135] https://github.com/STEllAR-GROUP/hpx/issues/585
[3136] https://github.com/STEllAR-GROUP/hpx/issues/584
[3137] https://github.com/STEllAR-GROUP/hpx/issues/581
[3138] https://github.com/STEllAR-GROUP/hpx/issues/580
[3139] https://github.com/STEllAR-GROUP/hpx/issues/578
[3140] https://github.com/STEllAR-GROUP/hpx/issues/576
[3141] https://github.com/STEllAR-GROUP/hpx/issues/575
[3142] https://github.com/STEllAR-GROUP/hpx/issues/574
[3143] https://github.com/STEllAR-GROUP/hpx/issues/571
[3144] https://github.com/STEllAR-GROUP/hpx/issues/570
[3145] https://github.com/STEllAR-GROUP/hpx/issues/569
[3146] https://github.com/STEllAR-GROUP/hpx/issues/568
[3147] https://github.com/STEllAR-GROUP/hpx/issues/567

- Issue #565[3148] - HPX doesn't shutdown properly
- Issue #564[3149] - Partial preprocessing of new component creation interface
- Issue #563[3150] - Add `hpx::start`/`hpx::stop` to avoid blocking main thread
- Issue #562[3151] - All command line arguments swallowed by hpx
- Issue #561[3152] - Boost.Tuple is not move aware
- Issue #558[3153] - `boost::shared_ptr<>` style semantics/syntax for client classes
- Issue #556[3154] - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
- Issue #555[3155] - `BOOST_FORCEINLINE` does not name a type
- Issue #554[3156] - Possible race condition in thread `get_id()`
- Issue #552[3157] - Move enable client_base
- Issue #550[3158] - Add stack size category 'huge'
- Issue #549[3159] - ShenEOS run seg-faults on single or distributed runs
- Issue #545[3160] - `AUTOGLOB` broken for add_hpx_component
- Issue #542[3161] - FindHPX_HDF5 still searches multiple times
- Issue #541[3162] - Quotes around application name in hpx::init
- Issue #539[3163] - Race conditition occuring with new lightweight threads
- Issue #535[3164] - hpx_run_tests.py exits with no error code when tests are missing
- Issue #530[3165] - Thread description(<unknown>) in logs
- Issue #523[3166] - Make thread objects more lightweight
- Issue #521[3167] - `hpx::error_code` is not usable for lightweight error handling
- Issue #520[3168] - Add full user environment to HPX logs
- Issue #519[3169] - Build succeeds, running fails
- Issue #517[3170] - Add a guard page to linux coroutine stacks

3148 https://github.com/STEllAR-GROUP/hpx/issues/565
3149 https://github.com/STEllAR-GROUP/hpx/issues/564
3150 https://github.com/STEllAR-GROUP/hpx/issues/563
3151 https://github.com/STEllAR-GROUP/hpx/issues/562
3152 https://github.com/STEllAR-GROUP/hpx/issues/561
3153 https://github.com/STEllAR-GROUP/hpx/issues/558
3154 https://github.com/STEllAR-GROUP/hpx/issues/556
3155 https://github.com/STEllAR-GROUP/hpx/issues/555
3156 https://github.com/STEllAR-GROUP/hpx/issues/554
3157 https://github.com/STEllAR-GROUP/hpx/issues/552
3158 https://github.com/STEllAR-GROUP/hpx/issues/550
3159 https://github.com/STEllAR-GROUP/hpx/issues/549
3160 https://github.com/STEllAR-GROUP/hpx/issues/545
3161 https://github.com/STEllAR-GROUP/hpx/issues/542
3162 https://github.com/STEllAR-GROUP/hpx/issues/541
3163 https://github.com/STEllAR-GROUP/hpx/issues/539
3164 https://github.com/STEllAR-GROUP/hpx/issues/535
3165 https://github.com/STEllAR-GROUP/hpx/issues/530
3166 https://github.com/STEllAR-GROUP/hpx/issues/523
3167 https://github.com/STEllAR-GROUP/hpx/issues/521
3168 https://github.com/STEllAR-GROUP/hpx/issues/520
3169 https://github.com/STEllAR-GROUP/hpx/issues/519
3170 https://github.com/STEllAR-GROUP/hpx/issues/517

- Issue #516[3171] - hpx::thread::detach suspends while holding locks, leads to hang in debug
- Issue #514[3172] - Preprocessed headers for <hpx/apply.hpp> don't compile
- Issue #513[3173] - Buildbot configuration problem
- Issue #512[3174] - Implement action based stack size customization
- Issue #511[3175] - Move action priority into a separate type trait
- Issue #510[3176] - trunk broken
- Issue #507[3177] - no matching function for call to `boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(h`
- Issue #505[3178] - undefined_symbol regression test currently failing
- Issue #502[3179] - Adding OpenCL and OCLM support to HPX for Windows and Linux
- Issue #501[3180] - find_package(HPX) sets cmake output variables
- Issue #500[3181] - wait_any/wait_all are badly named
- Issue #499[3182] - Add support for disabling pbs support in pbs runs
- Issue #498[3183] - Error during no-cache runs
- Issue #496[3184] - Add partial preprocessing support to cmake
- Issue #495[3185] - Support HPX modules exporting startup/shutdown functions only
- Issue #494[3186] - Allow modules to specify when to run startup/shutdown functions
- Issue #493[3187] - Avoid constructing a string in make_success_code
- Issue #492[3188] - Performance counter creation is no longer synchronized at startup
- Issue #491[3189] - Performance counter creation is no longer synchronized at startup
- Issue #490[3190] - Sheneos on_completed_bulk seg fault in distributed
- Issue #489[3191] - compiling issue with g++44
- Issue #488[3192] - Adding OpenCL and OCLM support to HPX for the MSVC platform
- Issue #487[3193] - FindHPX.cmake problems

[3171] https://github.com/STEllAR-GROUP/hpx/issues/516
[3172] https://github.com/STEllAR-GROUP/hpx/issues/514
[3173] https://github.com/STEllAR-GROUP/hpx/issues/513
[3174] https://github.com/STEllAR-GROUP/hpx/issues/512
[3175] https://github.com/STEllAR-GROUP/hpx/issues/511
[3176] https://github.com/STEllAR-GROUP/hpx/issues/510
[3177] https://github.com/STEllAR-GROUP/hpx/issues/507
[3178] https://github.com/STEllAR-GROUP/hpx/issues/505
[3179] https://github.com/STEllAR-GROUP/hpx/issues/502
[3180] https://github.com/STEllAR-GROUP/hpx/issues/501
[3181] https://github.com/STEllAR-GROUP/hpx/issues/500
[3182] https://github.com/STEllAR-GROUP/hpx/issues/499
[3183] https://github.com/STEllAR-GROUP/hpx/issues/498
[3184] https://github.com/STEllAR-GROUP/hpx/issues/496
[3185] https://github.com/STEllAR-GROUP/hpx/issues/495
[3186] https://github.com/STEllAR-GROUP/hpx/issues/494
[3187] https://github.com/STEllAR-GROUP/hpx/issues/493
[3188] https://github.com/STEllAR-GROUP/hpx/issues/492
[3189] https://github.com/STEllAR-GROUP/hpx/issues/491
[3190] https://github.com/STEllAR-GROUP/hpx/issues/490
[3191] https://github.com/STEllAR-GROUP/hpx/issues/489
[3192] https://github.com/STEllAR-GROUP/hpx/issues/488
[3193] https://github.com/STEllAR-GROUP/hpx/issues/487

- Issue #485[3194] - Change distributing_factory and binpacking_factory to use bulk creation
- Issue #484[3195] - Change `HPX_DONT_USE_PREPROCESSED_FILES` to `HPX_USE_PREPROCESSED_FILES`
- Issue #483[3196] - Memory counter for Windows
- Issue #479[3197] - strange errors appear when requesting performance counters on multiple nodes
- Issue #477[3198] - Create (global) timer for multi-threaded measurements
- Issue #472[3199] - Add partial preprocessing using Wave
- Issue #471[3200] - Segfault stack traces don't show up in release
- Issue #468[3201] - External projects need to link with internal components
- Issue #462[3202] - Startup/shutdown functions are called more than once
- Issue #458[3203] - Consolidate hpx::util::high_resolution_timer and `hpx::util::high_resolution_clock`
- Issue #457[3204] - index out of bounds in `allgather_and_gate` on 4 cores or more
- Issue #448[3205] - Make HPX compile with clang
- Issue #447[3206] - 'make tests' should execute tests on local installation
- Issue #446[3207] - Remove SVN-related code from the codebase
- Issue #444[3208] - race condition in smp
- Issue #441[3209] - Patched Boost.Serialization headers should only be installed if needed
- Issue #439[3210] - Components using `HPX_REGISTER_STARTUP_MODULE` fail to compile with MSVC
- Issue #436[3211] - Verify that no locks are being held while threads are suspended
- Issue #435[3212] - Installing HPX should not clobber existing Boost installation
- Issue #434[3213] - Logging external component failed (Boost 1.50)
- Issue #433[3214] - Runtime crash when building all examples
- Issue #432[3215] - Dataflow hangs on 512 cores/64 nodes
- Issue #430[3216] - Problem with distributing factory

3194 https://github.com/STEllAR-GROUP/hpx/issues/485
3195 https://github.com/STEllAR-GROUP/hpx/issues/484
3196 https://github.com/STEllAR-GROUP/hpx/issues/483
3197 https://github.com/STEllAR-GROUP/hpx/issues/479
3198 https://github.com/STEllAR-GROUP/hpx/issues/477
3199 https://github.com/STEllAR-GROUP/hpx/issues/472
3200 https://github.com/STEllAR-GROUP/hpx/issues/471
3201 https://github.com/STEllAR-GROUP/hpx/issues/468
3202 https://github.com/STEllAR-GROUP/hpx/issues/462
3203 https://github.com/STEllAR-GROUP/hpx/issues/458
3204 https://github.com/STEllAR-GROUP/hpx/issues/457
3205 https://github.com/STEllAR-GROUP/hpx/issues/448
3206 https://github.com/STEllAR-GROUP/hpx/issues/447
3207 https://github.com/STEllAR-GROUP/hpx/issues/446
3208 https://github.com/STEllAR-GROUP/hpx/issues/444
3209 https://github.com/STEllAR-GROUP/hpx/issues/441
3210 https://github.com/STEllAR-GROUP/hpx/issues/439
3211 https://github.com/STEllAR-GROUP/hpx/issues/436
3212 https://github.com/STEllAR-GROUP/hpx/issues/435
3213 https://github.com/STEllAR-GROUP/hpx/issues/434
3214 https://github.com/STEllAR-GROUP/hpx/issues/433
3215 https://github.com/STEllAR-GROUP/hpx/issues/432
3216 https://github.com/STEllAR-GROUP/hpx/issues/430

- Issue #424[3217] - File paths referring to XSL-files need to be properly escaped
- Issue #417[3218] - Make dataflow LCOs work out of the box by using partial preprocessing
- Issue #413[3219] - hpx_svnversion.py fails on Windows
- Issue #412[3220] - Make hpx::error_code equivalent to hpx::exception
- Issue #398[3221] - HPX clobbers out-of-tree application specific CMake variables (specifically `CMAKE_BUILD_TYPE`)
- Issue #394[3222] - Remove code generating random port numbers for network
- Issue #378[3223] - ShenEOS scaling issues
- Issue #354[3224] - Create a coroutines wrapper for Boost.Context
- Issue #349[3225] - Commandline option `--localities=N/-lN` should be necessary only on AGAS locality
- Issue #334[3226] - Add auto_index support to cmake based documentation toolchain
- Issue #318[3227] - Network benchmarks
- Issue #317[3228] - Implement network performance counters
- Issue #310[3229] - Duplicate logging entries
- Issue #230[3230] - Add compile time option to disable thread debugging info
- Issue #171[3231] - Add an INI option to turn off deadlock detection independently of logging
- Issue #170[3232] - OSHL internal counters are incorrect
- Issue #103[3233] - Better diagnostics for multiple component/action registerations under the same name
- Issue #48[3234] - Support for Darwin (Xcode + Clang)
- Issue #21[3235] - Build fails with GCC 4.6

2.10.14 *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.).

3217 https://github.com/STEllAR-GROUP/hpx/issues/424
3218 https://github.com/STEllAR-GROUP/hpx/issues/417
3219 https://github.com/STEllAR-GROUP/hpx/issues/413
3220 https://github.com/STEllAR-GROUP/hpx/issues/412
3221 https://github.com/STEllAR-GROUP/hpx/issues/398
3222 https://github.com/STEllAR-GROUP/hpx/issues/394
3223 https://github.com/STEllAR-GROUP/hpx/issues/378
3224 https://github.com/STEllAR-GROUP/hpx/issues/354
3225 https://github.com/STEllAR-GROUP/hpx/issues/349
3226 https://github.com/STEllAR-GROUP/hpx/issues/334
3227 https://github.com/STEllAR-GROUP/hpx/issues/318
3228 https://github.com/STEllAR-GROUP/hpx/issues/317
3229 https://github.com/STEllAR-GROUP/hpx/issues/310
3230 https://github.com/STEllAR-GROUP/hpx/issues/230
3231 https://github.com/STEllAR-GROUP/hpx/issues/171
3232 https://github.com/STEllAR-GROUP/hpx/issues/170
3233 https://github.com/STEllAR-GROUP/hpx/issues/103
3234 https://github.com/STEllAR-GROUP/hpx/issues/48
3235 https://github.com/STEllAR-GROUP/hpx/issues/21

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[3236].
- *HPX* now exposes `hpx::thread` which is compliant to the C++11 std::thread type except that it (purely locally) represents an *HPX* thread. This new type does not expose any of the remote capabilities of the underlying *HPX*-thread implementation.
- The type `hpx::lcos::future` is now compliant to the C++11 std::future<> type. This type can be used to synchronize both, local and remote operations. In both cases the control flow will 'return' to the future in order to trigger any continuation.
- The types `hpx::lcos::local::promise` and `hpx::lcos::local::packaged_task` are now compliant to the C++11 `std::promise<>` and `std::packaged_task<>` types. These can be used to create a future representing local work only. Use the types `hpx::lcos::promise` and `hpx::lcos::packaged_action` to wrap any (possibly remote) action into a future.
- `hpx::thread` and `hpx::lcos::future` are now cancelable.
- Added support for sequential and logic composition of `hpx::lcos::futures`. The member function `hpx::lcos::future::when` permits futures to be sequentially composed. The helper functions `hpx::wait_all`, `hpx::wait_any`, and `hpx::wait_n` can be used to wait for more than one future at a time.
- *HPX* now exposes `hpx::apply` and `hpx::async` as the preferred way of creating (or invoking) any deferred work. These functions are usable with various types of functions, function objects, and actions and provide a uniform way to spawn deferred tasks.
- *HPX* now utilizes `hpx::util::bind` to (partially) bind local functions and function objects, and also actions. Remote bound actions can have placeholders as well.
- *HPX* continuations are now fully polymorphic. The class `hpx::actions::forwarding_continuation` is an example of how the user can write is own types of continuations. It can be used to execute any function as an continuation of a particular action.
- Reworked the action invocation API to be fully conformant to normal functions. Actions can now be invoked using `hpx::apply`, `hpx::async`, or using the `operator()` implemented on actions. Actions themselves can now be cheaply instantiated as they do not have any members anymore.
- Reworked the lazy action invocation API. Actions can now be directly bound using `hpx::util::bind` by passing an action instance as the first argument.
- A minimal *HPX* program now looks like this:

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

int hpx_main()
{
    return hpx::finalize();
}

int main()
{
    return hpx::init();
}
```

[3236] http://www.open-std.org/jtc1/sc22/wg21

This removes the immediate dependency on the Boost.Program Options[3237] 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[3238], the minimal program can be written as:

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

// The arguments for hpx_main can be left off, which very similar to the
// behavior of ``main()`` as defined by C++.
int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

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

- Added performance counters exposing the number of component instances which are alive on a given locality.
- Added performance counters exposing then number of messages sent and received, the number of parcels sent and received, the number of bytes sent and received, the overall time required to send and receive data, and the overall time required to serialize and deserialize the data.
- Added a new component: `hpx::components::binpacking_factory` which is equivalent to the existing `hpx::components::distributing_factory` component, except that it equalizes the overall population of the components to create. It exposes two factory methods, one based on the number of existing instances of the component type to create, and one based on an arbitrary performance counter which will be queried for all relevant localities.
- Added API functions allowing to access elements of the diagnostic information embedded in the given exception: `hpx::get_locality_id`, `hpx::get_host_name`, `hpx::get_process_id`, `hpx::get_function_name`, `hpx::get_file_name`, `hpx::get_line_number`, `hpx::get_os_thread`, `hpx::get_thread_id`, and `hpx::get_thread_description`.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #71[3239] - GIDs that are not serialized via `handle_gid<>` should raise an exception
- Issue #105[3240] - Allow for `hpx::util::functions` to be registered in the AGAS symbolic namespace
- Issue #107[3241] - Nasty threadmanger race condition (reproducible in sheneos_test)
- Issue #108[3242] - Add millisecond resolution to *HPX* logs on Linux

3237 https://www.boost.org/doc/html/program_options.html
3238 https://www.boost.org/doc/html/program_options.html
3239 https://github.com/STEllAR-GROUP/hpx/issues/71
3240 https://github.com/STEllAR-GROUP/hpx/issues/105
3241 https://github.com/STEllAR-GROUP/hpx/issues/107
3242 https://github.com/STEllAR-GROUP/hpx/issues/108

- Issue #110[3243] - Shutdown hang in distributed with release build
- Issue #116[3244] - Don't use TSS for the applier and runtime pointers
- Issue #162[3245] - Move local synchronous execution shortcut from hpx::function to the applier
- Issue #172[3246] - Cache sources in CMake and check if they change manually
- Issue #178[3247] - Add an INI option to turn off ranged-based AGAS caching
- Issue #187[3248] - Support for disabling performance counter deployment
- Issue #202[3249] - Support for sending performance counter data to a specific file
- Issue #218[3250] - boost.coroutines allows different stack sizes, but stack pool is unaware of this
- Issue #231[3251] - Implement movable `boost::bind`
- Issue #232[3252] - Implement movable `boost::function`
- Issue #236[3253] - Allow binding `hpx::util::function` to actions
- Issue #239[3254] - Replace `hpx::function` with `hpx::util::function`
- Issue #240[3255] - Can't specify RemoteResult with lcos::async
- Issue #242[3256] - REGISTER_TEMPLATE support for plain actions
- Issue #243[3257] - `handle_gid<>` support for `hpx::util::function`
- Issue #245[3258] - `*_c_cache code` throws an exception if the queried GID is not in the local cache
- Issue #246[3259] - Undefined references in dataflow/adaptive1d example
- Issue #252[3260] - Problems configuring sheneos with CMake
- Issue #254[3261] - Lifetime of components doesn't end when client goes out of scope
- Issue #259[3262] - CMake does not detect that MSVC10 has lambdas
- Issue #260[3263] - io_service_pool segfault
- Issue #261[3264] - Late parcel executed outside of pxthread
- Issue #263[3265] - Cannot select allocator with CMake

[3243] https://github.com/STEllAR-GROUP/hpx/issues/110
[3244] https://github.com/STEllAR-GROUP/hpx/issues/116
[3245] https://github.com/STEllAR-GROUP/hpx/issues/162
[3246] https://github.com/STEllAR-GROUP/hpx/issues/172
[3247] https://github.com/STEllAR-GROUP/hpx/issues/178
[3248] https://github.com/STEllAR-GROUP/hpx/issues/187
[3249] https://github.com/STEllAR-GROUP/hpx/issues/202
[3250] https://github.com/STEllAR-GROUP/hpx/issues/218
[3251] https://github.com/STEllAR-GROUP/hpx/issues/231
[3252] https://github.com/STEllAR-GROUP/hpx/issues/232
[3253] https://github.com/STEllAR-GROUP/hpx/issues/236
[3254] https://github.com/STEllAR-GROUP/hpx/issues/239
[3255] https://github.com/STEllAR-GROUP/hpx/issues/240
[3256] https://github.com/STEllAR-GROUP/hpx/issues/242
[3257] https://github.com/STEllAR-GROUP/hpx/issues/243
[3258] https://github.com/STEllAR-GROUP/hpx/issues/245
[3259] https://github.com/STEllAR-GROUP/hpx/issues/246
[3260] https://github.com/STEllAR-GROUP/hpx/issues/252
[3261] https://github.com/STEllAR-GROUP/hpx/issues/254
[3262] https://github.com/STEllAR-GROUP/hpx/issues/259
[3263] https://github.com/STEllAR-GROUP/hpx/issues/260
[3264] https://github.com/STEllAR-GROUP/hpx/issues/261
[3265] https://github.com/STEllAR-GROUP/hpx/issues/263

- Issue #264[3266] - Fix allocator select
- Issue #267[3267] - Runtime error for hello_world
- Issue #269[3268] - pthread_affinity_np test fails to compile
- Issue #270[3269] - Compiler noise due to -Wcast-qual
- Issue #275[3270] - Problem with configuration tests/include paths on Gentoo
- Issue #325[3271] - Sheneos is 200-400 times slower than the fortran equivalent
- Issue #331[3272] - `hpx::init` and `hpx_main()` should not depend on program_options
- Issue #333[3273] - Add doxygen support to CMake for doc toolchain
- Issue #340[3274] - Performance counters for parcels
- Issue #346[3275] - Component loading error when running hello_world in distributed on MSVC2010
- Issue #362[3276] - Missing initializer error
- Issue #363[3277] - Parcel port serialization error
- Issue #366[3278] - Parcel buffering leads to types incompatible exception
- Issue #368[3279] - Scalable alternative to rand() needed for *HPX*
- Issue #369[3280] - IB over IP is substantially slower than just using standard TCP/IP
- Issue #374[3281] - `hpx::lcos::wait` should work with dataflows and arbitrary classes meeting the future interface
- Issue #375[3282] - Conflicting/ambiguous overloads of `hpx::lcos::wait`
- Issue #376[3283] - Find_HPX.cmake should set CMake variable HPX_FOUND for out of tree builds
- Issue #377[3284] - ShenEOS interpolate bulk and interpolate_one_bulk are broken
- Issue #379[3285] - Add support for distributed runs under SLURM
- Issue #382[3286] - _Unwind_Word not declared in boost.backtrace
- Issue #387[3287] - Doxygen should look only at list of specified files
- Issue #388[3288] - Running `make install` on an out-of-tree application is broken

[3266] https://github.com/STEllAR-GROUP/hpx/issues/264
[3267] https://github.com/STEllAR-GROUP/hpx/issues/267
[3268] https://github.com/STEllAR-GROUP/hpx/issues/269
[3269] https://github.com/STEllAR-GROUP/hpx/issues/270
[3270] https://github.com/STEllAR-GROUP/hpx/issues/275
[3271] https://github.com/STEllAR-GROUP/hpx/issues/325
[3272] https://github.com/STEllAR-GROUP/hpx/issues/331
[3273] https://github.com/STEllAR-GROUP/hpx/issues/333
[3274] https://github.com/STEllAR-GROUP/hpx/issues/340
[3275] https://github.com/STEllAR-GROUP/hpx/issues/346
[3276] https://github.com/STEllAR-GROUP/hpx/issues/362
[3277] https://github.com/STEllAR-GROUP/hpx/issues/363
[3278] https://github.com/STEllAR-GROUP/hpx/issues/366
[3279] https://github.com/STEllAR-GROUP/hpx/issues/368
[3280] https://github.com/STEllAR-GROUP/hpx/issues/369
[3281] https://github.com/STEllAR-GROUP/hpx/issues/374
[3282] https://github.com/STEllAR-GROUP/hpx/issues/375
[3283] https://github.com/STEllAR-GROUP/hpx/issues/376
[3284] https://github.com/STEllAR-GROUP/hpx/issues/377
[3285] https://github.com/STEllAR-GROUP/hpx/issues/379
[3286] https://github.com/STEllAR-GROUP/hpx/issues/382
[3287] https://github.com/STEllAR-GROUP/hpx/issues/387
[3288] https://github.com/STEllAR-GROUP/hpx/issues/388

- Issue #391[3289] - Out-of-tree application segfaults when running in qsub
- Issue #392[3290] - Remove HPX_NO_INSTALL option from cmake build system
- Issue #396[3291] - Pragma related warnings when compiling with older gcc versions
- Issue #399[3292] - Out of tree component build problems
- Issue #400[3293] - Out of source builds on Windows: linker should not receive compiler flags
- Issue #401[3294] - Out of source builds on Windows: components need to be linked with hpx_serialization
- Issue #404[3295] - gfortran fails to link automatically when fortran files are present
- Issue #405[3296] - Inability to specify linking order for external libraries
- Issue #406[3297] - Adapt action limits such that dataflow applications work without additional defines
- Issue #415[3298] - `locality_results` is not a member of `hpx::components::server`
- Issue #425[3299] - Breaking changes to `traits::*result` wrt `std::vector<id_type>`
- Issue #426[3300] - AUTOGLOB needs to be updated to support fortran

2.10.15 *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[3301] - Don't require install path to be known at compile time.
- Issue #371[3302] - Add hpx iostreams to standard build.
- Issue #384[3303] - Fix compilation with GCC 4.7.
- Issue #390[3304] - Remove keep_factory_alive startup call from ShenEOS; add shutdown call to H5close.
- Issue #393[3305] - Thread affinity control is broken.

[3289] https://github.com/STEllAR-GROUP/hpx/issues/391
[3290] https://github.com/STEllAR-GROUP/hpx/issues/392
[3291] https://github.com/STEllAR-GROUP/hpx/issues/396
[3292] https://github.com/STEllAR-GROUP/hpx/issues/399
[3293] https://github.com/STEllAR-GROUP/hpx/issues/400
[3294] https://github.com/STEllAR-GROUP/hpx/issues/401
[3295] https://github.com/STEllAR-GROUP/hpx/issues/404
[3296] https://github.com/STEllAR-GROUP/hpx/issues/405
[3297] https://github.com/STEllAR-GROUP/hpx/issues/406
[3298] https://github.com/STEllAR-GROUP/hpx/issues/415
[3299] https://github.com/STEllAR-GROUP/hpx/issues/425
[3300] https://github.com/STEllAR-GROUP/hpx/issues/426
[3301] https://github.com/STEllAR-GROUP/hpx/issues/295
[3302] https://github.com/STEllAR-GROUP/hpx/issues/371
[3303] https://github.com/STEllAR-GROUP/hpx/issues/384
[3304] https://github.com/STEllAR-GROUP/hpx/issues/390
[3305] https://github.com/STEllAR-GROUP/hpx/issues/393

Bug fixes (commits)

Here is a list of the important commits included in this point release:

- r7642 - External: Fix backtrace memory violation.
- **r7775 - Components: Fix symbol visibility bug with component startup** providers. This prevents one components providers from overriding another components.
- r7778 - Components: Fix startup/shutdown provider shadowing issues.

2.10.16 *HPX* V0.8.0 (Mar 23, 2012)

We have had roughly 1000 commits since the last release and we have closed approximately 70 tickets (bugs, feature requests, etc.).

General changes

- Improved PBS support, allowing for arbitrary naming schemes of node-hostnames.
- Finished verification of the reference counting framework.
- Implemented decrement merging logic to optimize the distributed reference counting system.
- Restructured the LCO framework. Renamed `hpx::lcos::eager_future<>` and `hpx::lcos::lazy_future<>` into `hpx::lcos::packaged_task` and `hpx::lcos::deferred_packaged_task`. Split `hpx::lcos::promise` into `hpx::lcos::packaged_task` and `hpx::lcos::future`. Added 'local' futures (in namespace `hpx::lcos::local`).
- Improved the general performance of local and remote action invocations. This (under certain circumstances) drastically reduces the number of copies created for each of the parameters and return values.
- 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[3306] (thanks to Matthew Anderson for contributing this application).
- GTC (Gyrokinetic Toroidal Code)[3307]: a skeleton for particle in cell type codes.
- Random Memory Access: an example demonstrating random memory accesses in a large array
- ShenEOS example[3308], demonstrating partitioning of large read-only data structures and exposing an interpolation API.
- Sine performance counter demo.
- Accumulator examples demonstrating how to write and use *HPX* components.
- Quickstart examples (like hello_world, fibonacci, quicksort, factorial, etc.) demonstrating simple *HPX* concepts which introduce some of the concepts in *HPX*.
- Load balancing and work stealing demos.

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`.

3306 http://www.graph500.org/
3307 http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/gtc/
3308 http://stellarcollapse.org/equationofstate

- 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[3309] - Specialize handle_gid<> for examples and tests
- Issue #72[3310] - Fix AGAS reference counting
- Issue #104[3311] - heartbeat throws an exception when decrefing the performance counter it's watching
- Issue #111[3312] - throttle causes an exception on the target application
- Issue #142[3313] - One failed component loading causes an unrelated component to fail
- Issue #165[3314] - Remote exception propagation bug in AGAS reference counting test
- Issue #186[3315] - Test credit exhaustion/splitting (e.g. prepare_gid and symbol NS)
- Issue #188[3316] - Implement remaining AGAS reference counting test cases
- Issue #258[3317] - No type checking of GIDs in stubs classes
- Issue #271[3318] - Seg fault/shared pointer assertion in distributed code
- Issue #281[3319] - CMake options need descriptive text
- Issue #283[3320] - AGAS caching broken (gva_cache needs to be rewritten with ICL)
- Issue #285[3321] - HPX_INSTALL root directory not the same as CMAKE_INSTALL_PREFIX
- Issue #286[3322] - New segfault in dataflow applications
- Issue #289[3323] - Exceptions should only be logged if not handled
- Issue #290[3324] - c++11 tests failure
- Issue #293[3325] - Build target for component libraries
- Issue #296[3326] - Compilation error with Boost V1.49rc1
- Issue #298[3327] - Illegal instructions on termination
- Issue #299[3328] - gravity aborts with multiple threads

[3309] https://github.com/STEllAR-GROUP/hpx/issues/31
[3310] https://github.com/STEllAR-GROUP/hpx/issues/72
[3311] https://github.com/STEllAR-GROUP/hpx/issues/104
[3312] https://github.com/STEllAR-GROUP/hpx/issues/111
[3313] https://github.com/STEllAR-GROUP/hpx/issues/142
[3314] https://github.com/STEllAR-GROUP/hpx/issues/165
[3315] https://github.com/STEllAR-GROUP/hpx/issues/186
[3316] https://github.com/STEllAR-GROUP/hpx/issues/188
[3317] https://github.com/STEllAR-GROUP/hpx/issues/258
[3318] https://github.com/STEllAR-GROUP/hpx/issues/271
[3319] https://github.com/STEllAR-GROUP/hpx/issues/281
[3320] https://github.com/STEllAR-GROUP/hpx/issues/283
[3321] https://github.com/STEllAR-GROUP/hpx/issues/285
[3322] https://github.com/STEllAR-GROUP/hpx/issues/286
[3323] https://github.com/STEllAR-GROUP/hpx/issues/289
[3324] https://github.com/STEllAR-GROUP/hpx/issues/290
[3325] https://github.com/STEllAR-GROUP/hpx/issues/293
[3326] https://github.com/STEllAR-GROUP/hpx/issues/296
[3327] https://github.com/STEllAR-GROUP/hpx/issues/298
[3328] https://github.com/STEllAR-GROUP/hpx/issues/299

- Issue #301[3329] - Build error with Boost trunk
- Issue #303[3330] - Logging assertion failure in distributed runs
- Issue #304[3331] - Exception 'what' strings are lost when exceptions from decode_parcel are reported
- Issue #306[3332] - Performance counter user interface issues
- Issue #307[3333] - Logging exception in distributed runs
- Issue #308[3334] - Logging deadlocks in distributed
- Issue #309[3335] - Reference counting test failures and exceptions
- Issue #311[3336] - Merge AGAS remote_interface with the runtime_support object
- Issue #314[3337] - Object tracking for id_types
- Issue #315[3338] - Remove handle_gid and handle credit splitting in id_type serialization
- Issue #320[3339] - applier::get_locality_id() should return an error value (or throw an exception)
- Issue #321[3340] - Optimization for id_types which are never split should be restored
- Issue #322[3341] - Command line processing ignored with Boost 1.47.0
- Issue #323[3342] - Credit exhaustion causes object to stay alive
- Issue #324[3343] - Duplicate exception messages
- Issue #326[3344] - Integrate Quickbook with CMake
- Issue #329[3345] - –help and –version should still work
- Issue #330[3346] - Create pkg-config files
- Issue #337[3347] - Improve usability of performance counter timestamps
- Issue #338[3348] - Non-std exceptions deriving from std::exceptions in tfunc may be sliced
- Issue #339[3349] - Decrease the number of send_pending_parcels threads
- Issue #343[3350] - Dynamically setting the stack size doesn't work
- Issue #351[3351] - 'make install' does not update documents

[3329] https://github.com/STEllAR-GROUP/hpx/issues/301
[3330] https://github.com/STEllAR-GROUP/hpx/issues/303
[3331] https://github.com/STEllAR-GROUP/hpx/issues/304
[3332] https://github.com/STEllAR-GROUP/hpx/issues/306
[3333] https://github.com/STEllAR-GROUP/hpx/issues/307
[3334] https://github.com/STEllAR-GROUP/hpx/issues/308
[3335] https://github.com/STEllAR-GROUP/hpx/issues/309
[3336] https://github.com/STEllAR-GROUP/hpx/issues/311
[3337] https://github.com/STEllAR-GROUP/hpx/issues/314
[3338] https://github.com/STEllAR-GROUP/hpx/issues/315
[3339] https://github.com/STEllAR-GROUP/hpx/issues/320
[3340] https://github.com/STEllAR-GROUP/hpx/issues/321
[3341] https://github.com/STEllAR-GROUP/hpx/issues/322
[3342] https://github.com/STEllAR-GROUP/hpx/issues/323
[3343] https://github.com/STEllAR-GROUP/hpx/issues/324
[3344] https://github.com/STEllAR-GROUP/hpx/issues/326
[3345] https://github.com/STEllAR-GROUP/hpx/issues/329
[3346] https://github.com/STEllAR-GROUP/hpx/issues/330
[3347] https://github.com/STEllAR-GROUP/hpx/issues/337
[3348] https://github.com/STEllAR-GROUP/hpx/issues/338
[3349] https://github.com/STEllAR-GROUP/hpx/issues/339
[3350] https://github.com/STEllAR-GROUP/hpx/issues/343
[3351] https://github.com/STEllAR-GROUP/hpx/issues/351

- Issue #353[3352] - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs
- Issue #355[3353] - 'make' doesn't do anything after correct configuration
- Issue #356[3354] - Don't use `hpx::util::static_` in topology code
- Issue #359[3355] - Infinite recursion in hpx::tuple serialization
- Issue #361[3356] - Add compile time option to disable logging completely
- Issue #364[3357] - Installation seriously broken in r7443

2.10.17 *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.).

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.

[3352] https://github.com/STEllAR-GROUP/hpx/issues/353
[3353] https://github.com/STEllAR-GROUP/hpx/issues/355
[3354] https://github.com/STEllAR-GROUP/hpx/issues/356
[3355] https://github.com/STEllAR-GROUP/hpx/issues/359
[3356] https://github.com/STEllAR-GROUP/hpx/issues/361
[3357] https://github.com/STEllAR-GROUP/hpx/issues/364

- 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[3358] - Integrate Windows/Linux CMake code for *HPX* core
- Issue #32[3359] - hpx::cout() should be hpx::cout
- Issue #33[3360] - AGAS V2 legacy client does not properly handle error_code
- Issue #60[3361] - AGAS: allow for registerid to optionally take ownership of the gid
- Issue #62[3362] - adaptive1d compilation failure in Fusion
- Issue #64[3363] - Parcel subsystem doesn't resolve domain names
- Issue #83[3364] - No error handling if no console is available
- Issue #84[3365] - No error handling if a hosted locality is treated as the bootstrap server
- Issue #90[3366] - Add general commandline option -N
- Issue #91[3367] - Add possibility to read command line arguments from file

[3358] https://github.com/STEllAR-GROUP/hpx/issues/28
[3359] https://github.com/STEllAR-GROUP/hpx/issues/32
[3360] https://github.com/STEllAR-GROUP/hpx/issues/33
[3361] https://github.com/STEllAR-GROUP/hpx/issues/60
[3362] https://github.com/STEllAR-GROUP/hpx/issues/62
[3363] https://github.com/STEllAR-GROUP/hpx/issues/64
[3364] https://github.com/STEllAR-GROUP/hpx/issues/83
[3365] https://github.com/STEllAR-GROUP/hpx/issues/84
[3366] https://github.com/STEllAR-GROUP/hpx/issues/90
[3367] https://github.com/STEllAR-GROUP/hpx/issues/91

- Issue #92[3368] - Always log exceptions/errors to the log file
- Issue #93[3369] - Log the command line/program name
- Issue #95[3370] - Support for distributed launches
- Issue #97[3371] - Attempt to create a bad component type in AMR examples
- Issue #100[3372] - factorial and factorial_get examples trigger AGAS component type assertions
- Issue #101[3373] - Segfault when hpx::process::here() is called in fibonacci2
- Issue #102[3374] - unknown_component_address in int_object_semaphore_client
- Issue #114[3375] - marduk raises assertion with default parameters
- Issue #115[3376] - Logging messages for SMP runs (on the console) shouldn't be buffered
- Issue #119[3377] - marduk linking strategy breaks other applications
- Issue #121[3378] - pbsdsh problem
- Issue #123[3379] - marduk, dataflow and adaptive1d fail to build
- Issue #124[3380] - Lower default preprocessing arity
- Issue #125[3381] - Move hpx::detail::diagnostic_information out of the detail namespace
- Issue #126[3382] - Test definitions for AGAS reference counting
- Issue #128[3383] - Add averaging performance counter
- Issue #129[3384] - Error with endian.hpp while building adaptive1d
- Issue #130[3385] - Bad initialization of performance counters
- Issue #131[3386] - Add global startup/shutdown functions to component modules
- Issue #132[3387] - Avoid using auto_ptr
- Issue #133[3388] - On Windows hpx.dll doesn't get installed
- Issue #134[3389] - HPX_LIBRARY does not reflect real library name (on Windows)
- Issue #135[3390] - Add detection of unique_ptr to build system

[3368] https://github.com/STEllAR-GROUP/hpx/issues/92
[3369] https://github.com/STEllAR-GROUP/hpx/issues/93
[3370] https://github.com/STEllAR-GROUP/hpx/issues/95
[3371] https://github.com/STEllAR-GROUP/hpx/issues/97
[3372] https://github.com/STEllAR-GROUP/hpx/issues/100
[3373] https://github.com/STEllAR-GROUP/hpx/issues/101
[3374] https://github.com/STEllAR-GROUP/hpx/issues/102
[3375] https://github.com/STEllAR-GROUP/hpx/issues/114
[3376] https://github.com/STEllAR-GROUP/hpx/issues/115
[3377] https://github.com/STEllAR-GROUP/hpx/issues/119
[3378] https://github.com/STEllAR-GROUP/hpx/issues/121
[3379] https://github.com/STEllAR-GROUP/hpx/issues/123
[3380] https://github.com/STEllAR-GROUP/hpx/issues/124
[3381] https://github.com/STEllAR-GROUP/hpx/issues/125
[3382] https://github.com/STEllAR-GROUP/hpx/issues/126
[3383] https://github.com/STEllAR-GROUP/hpx/issues/128
[3384] https://github.com/STEllAR-GROUP/hpx/issues/129
[3385] https://github.com/STEllAR-GROUP/hpx/issues/130
[3386] https://github.com/STEllAR-GROUP/hpx/issues/131
[3387] https://github.com/STEllAR-GROUP/hpx/issues/132
[3388] https://github.com/STEllAR-GROUP/hpx/issues/133
[3389] https://github.com/STEllAR-GROUP/hpx/issues/134
[3390] https://github.com/STEllAR-GROUP/hpx/issues/135

- Issue #137[3391] - Add command line option allowing to repeatedly evaluate performance counters
- Issue #139[3392] - Logging is broken
- Issue #140[3393] - CMake problem on windows
- Issue #141[3394] - Move all non-component libraries into $PREFIX/lib/hpx
- Issue #143[3395] - adaptive1d throws an exception with the default command line options
- Issue #146[3396] - Early exception handling is broken
- Issue #147[3397] - Sheneos doesn't link on Linux
- Issue #149[3398] - sheneos_test hangs
- Issue #154[3399] - Compilation fails for r5661
- Issue #155[3400] - Sine performance counters example chokes on chrono headers
- Issue #156[3401] - Add build type to –version
- Issue #157[3402] - Extend AGAS caching to store gid ranges
- Issue #158[3403] - r5691 doesn't compile
- Issue #160[3404] - Re-add AGAS function for resolving a locality to its prefix
- Issue #168[3405] - Managed components should be able to access their own GID
- Issue #169[3406] - Rewrite AGAS future pool
- Issue #179[3407] - Complete switch to request class for AGAS server interface
- Issue #182[3408] - Sine performance counter is loaded by other examples
- Issue #185[3409] - Write tests for symbol namespace reference counting
- Issue #191[3410] - Assignment of read-only variable in point_geometry
- Issue #200[3411] - Seg faults when querying performance counters
- Issue #204[3412] - –ifnames and suffix stripping needs to be more generic
- Issue #205[3413] - –list-* and –print-counter-* options do not work together and produce no warning

[3391] https://github.com/STEllAR-GROUP/hpx/issues/137
[3392] https://github.com/STEllAR-GROUP/hpx/issues/139
[3393] https://github.com/STEllAR-GROUP/hpx/issues/140
[3394] https://github.com/STEllAR-GROUP/hpx/issues/141
[3395] https://github.com/STEllAR-GROUP/hpx/issues/143
[3396] https://github.com/STEllAR-GROUP/hpx/issues/146
[3397] https://github.com/STEllAR-GROUP/hpx/issues/147
[3398] https://github.com/STEllAR-GROUP/hpx/issues/149
[3399] https://github.com/STEllAR-GROUP/hpx/issues/154
[3400] https://github.com/STEllAR-GROUP/hpx/issues/155
[3401] https://github.com/STEllAR-GROUP/hpx/issues/156
[3402] https://github.com/STEllAR-GROUP/hpx/issues/157
[3403] https://github.com/STEllAR-GROUP/hpx/issues/158
[3404] https://github.com/STEllAR-GROUP/hpx/issues/160
[3405] https://github.com/STEllAR-GROUP/hpx/issues/168
[3406] https://github.com/STEllAR-GROUP/hpx/issues/169
[3407] https://github.com/STEllAR-GROUP/hpx/issues/179
[3408] https://github.com/STEllAR-GROUP/hpx/issues/182
[3409] https://github.com/STEllAR-GROUP/hpx/issues/185
[3410] https://github.com/STEllAR-GROUP/hpx/issues/191
[3411] https://github.com/STEllAR-GROUP/hpx/issues/200
[3412] https://github.com/STEllAR-GROUP/hpx/issues/204
[3413] https://github.com/STEllAR-GROUP/hpx/issues/205

- Issue #207[3414] - Implement decrement entry merging
- Issue #208[3415] - Replace the spinlocks in AGAS with hpx::lcos::local_mutexes
- Issue #210[3416] - Add an –ifprefix option
- Issue #214[3417] - Performance test for PX-thread creation
- Issue #216[3418] - VS2010 compilation
- Issue #222[3419] - r6045 context_linux_x86.hpp
- Issue #223[3420] - fibonacci hangs when changing the state of an active thread
- Issue #225[3421] - Active threads end up in the FEB wait queue
- Issue #226[3422] - VS Build Error for Accumulator Client
- Issue #228[3423] - Move all traits into namespace hpx::traits
- Issue #229[3424] - Invalid initialization of reference in thread_init_data
- Issue #235[3425] - Invalid GID in iostreams
- Issue #238[3426] - Demangle type names for the default implementation of get_action_name
- Issue #241[3427] - C++11 support breaks GCC 4.5
- Issue #247[3428] - Reference to temporary with GCC 4.4
- Issue #248[3429] - Seg fault at shutdown with GCC 4.4
- Issue #253[3430] - Default component action registration kills compiler
- Issue #272[3431] - G++ unrecognized command line option
- Issue #273[3432] - quicksort example doesn't compile
- Issue #277[3433] - Invalid CMake logic for Windows

[3414] https://github.com/STEllAR-GROUP/hpx/issues/207
[3415] https://github.com/STEllAR-GROUP/hpx/issues/208
[3416] https://github.com/STEllAR-GROUP/hpx/issues/210
[3417] https://github.com/STEllAR-GROUP/hpx/issues/214
[3418] https://github.com/STEllAR-GROUP/hpx/issues/216
[3419] https://github.com/STEllAR-GROUP/hpx/issues/222
[3420] https://github.com/STEllAR-GROUP/hpx/issues/223
[3421] https://github.com/STEllAR-GROUP/hpx/issues/225
[3422] https://github.com/STEllAR-GROUP/hpx/issues/226
[3423] https://github.com/STEllAR-GROUP/hpx/issues/228
[3424] https://github.com/STEllAR-GROUP/hpx/issues/229
[3425] https://github.com/STEllAR-GROUP/hpx/issues/235
[3426] https://github.com/STEllAR-GROUP/hpx/issues/238
[3427] https://github.com/STEllAR-GROUP/hpx/issues/241
[3428] https://github.com/STEllAR-GROUP/hpx/issues/247
[3429] https://github.com/STEllAR-GROUP/hpx/issues/248
[3430] https://github.com/STEllAR-GROUP/hpx/issues/253
[3431] https://github.com/STEllAR-GROUP/hpx/issues/272
[3432] https://github.com/STEllAR-GROUP/hpx/issues/273
[3433] https://github.com/STEllAR-GROUP/hpx/issues/277

2.11 About *HPX*

2.11.1 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)[3434], a multi-disciplinary research institute at Louisiana State University (LSU)[3435]. The ParalleX group was working to develop a new and experimental execution model for future high performance computing architectures. This model was christened ParalleX. The first implementations of ParalleX were crude, and many of those designs had to be discarded entirely. However, over time the team learned quite a bit about how to design a parallel, distributed runtime system which implements the concepts of ParalleX.

From the very beginning, this endeavour has been a group effort. In addition to a handful of interested researchers, there have always been graduate and undergraduate students participating in the discussions, design, and implementation of *HPX*. In 2011 we decided to formalize our collective research efforts by creating the STE||AR[3436] 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[3437] 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[3438] 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[3439] and ParalleX, see *People* and *Why HPX?*.

2.11.2 People

The STE||AR[3440] 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[3441] 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

[3434] https://www.cct.lsu.edu
[3435] https://www.lsu.edu
[3436] https://stellar-group.org
[3437] https://stellar-group.org
[3438] https://www.boost.org/
[3439] https://stellar-group.org
[3440] https://stellar-group.org
[3441] https://stellar-group.org

cores in a multi-core system. By extension, these application will not be able to exploit high-end exascale computing systems which are likely to employ hundreds of millions of such cores by the end of this decade.

Critical bottlenecks to the effective use of new generation high performance computing (HPC) systems include:

- *Starvation*: due to lack of usable application parallelism and means of managing it,
- *Overhead*: reduction to permit strong scalability, improve efficiency, and enable dynamic resource management,
- *Latency*: from remote access across system or to local memories,
- *Contention*: due to multicore chip I/O pins, memory banks, and system interconnects.

The ParalleX model has been devised to address these challenges by enabling a new computing dynamic through the application of message-driven computation in a global address space context with lightweight synchronization. The work on *HPX* is centered around implementing the concepts as defined by the ParalleX model. *HPX* is currently targeted at conventional machines, such as classical Linux based Beowulf clusters and SMP nodes.

We fully understand that the success of *HPX* (and ParalleX) is very much the result of the work of many people. To see a list of who is contributing see our tables below.

HPX contributors

Table 2.39: Contributors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT)[3442], Louisiana State University (LSU)[3443]	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture[3444], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[3445]	thom.heller@gmail.com
Agustin Berge	Center for Computation and Technology (CCT)[3446], Louisiana State University (LSU)[3447]	kaballo86@hotmail.com
Mikael Simberg	Swiss National Supercomputing Centre[3448]	simbergm@cscs.ch
John Biddiscombe	Swiss National Supercomputing Centre[3449]	biddisco@cscs.ch
Anton Bikineev	Center for Computation and Technology (CCT)[3450], Louisiana State University (LSU)[3451]	ant.bikineev@gmail.com
Martin Stumpf	Department of Computer Science 3 - Computer Architecture[3452], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[3453]	martin.h.stumpf@gmail.com
Bryce Adelstein Lelbach	NVIDIA[3454]	brycelelbach@gmail.com
Shuangyang Yang	Center for Computation and Technology (CCT)[3455], Louisiana State University (LSU)[3456]	syang16@cct.lsu.edu
Jeroen Habraken	Technische Universiteit Eindhoven[3457]	jhabraken@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT)[3458], Louisiana State University (LSU)[3459]	sbrandt@cct.lsu.edu
Antoine Tran Tan	Center for Computation and Technology (CCT)[3460], Louisiana State University (LSU)[3461]	antoine.trantan@lri.fr
Adrian Serio	Center for Computation and Technology (CCT)[3462], Louisiana State University (LSU)[3463]	aserio@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST)[3464], Indiana University (IU)[3465]	mbrodowi@indiana.edu

Contributors to this document

Table 2.40: Documentation authors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT)[3466], Louisiana State University (LSU)[3467]	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture[3468], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[3469]	thom.heller@gmail.com
Bryce Adelstein Lelbach	NVIDIA[3470]	brycelelbach@gmail.com
Vinay C Amatya	Center for Computation and Technology (CCT)[3471], Louisiana State University (LSU)[3472]	vamatya@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT)[3473], Louisiana State University (LSU)[3474]	sbrandt@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST)[3475], Indiana University (IU)[3476]	mbrodowi@indiana.edu
Adrian Serio	Center for Computation and Technology (CCT)[3477], Louisiana State University (LSU)[3478]	aserio@cct.lsu.edu

3442 https://www.cct.lsu.edu
3443 https://www.lsu.edu
3444 https://www3.cs.fau.de
3445 https://www.fau.de
3446 https://www.cct.lsu.edu
3447 https://www.lsu.edu
3448 https://www.cscs.ch
3449 https://www.cscs.ch
3450 https://www.cct.lsu.edu
3451 https://www.lsu.edu
3452 https://www3.cs.fau.de
3453 https://www.fau.de
3454 https://nvidia.com/
3455 https://www.cct.lsu.edu
3456 https://www.lsu.edu
3457 https://www.tui.nl
3458 https://www.cct.lsu.edu
3459 https://www.lsu.edu
3460 https://www.cct.lsu.edu
3461 https://www.lsu.edu
3462 https://www.cct.lsu.edu
3463 https://www.lsu.edu
3464 https://pti.iu.edu
3465 https://www.iu.edu
3466 https://www.cct.lsu.edu
3467 https://www.lsu.edu
3468 https://www3.cs.fau.de
3469 https://www.fau.de
3470 https://nvidia.com/
3471 https://www.cct.lsu.edu
3472 https://www.lsu.edu
3473 https://www.cct.lsu.edu
3474 https://www.lsu.edu
3475 https://pti.iu.edu
3476 https://www.iu.edu
3477 https://www.cct.lsu.edu
3478 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.

- Jakub Golinowski, for implementing an *HPX* backend for OpenCV and in the process improving documentation and reporting issues.
- Mikael Simberg (Swiss National Supercomputing Centre[3479]), 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.
- Zach Byerly (Louisiana State University (LSU)[3480]), 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)[3481]), 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)[3482]), 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.

[3479] https://www.cscs.ch
[3480] https://www.lsu.edu
[3481] https://www.lsu.edu
[3482] https://www.lsu.edu

- Bibek Wagle (Louisiana State University (LSU)[3483]), 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[3484]) with *HPX*. He reported various problems and submitted several patches to fix issues allowing for a better integration with LibGeoDecomp[3485].
- 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)[3486]), 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[3487] C/C++ package manager.
- Vinay C Amatya (Louisiana State University (LSU)[3488]), who contributed to the documentation and provided some of the *HPX* examples.
- Kevin Huck and Nick Chaimov (University of Oregon[3489]), who contributed the integration of APEX (Autonomic Performance Environment for eXascale) with *HPX*.
- Francisco Jose Tapia, who helped with implementing the parallel sort algorithm for *HPX*.
- Patrick Diehl, who worked on implementing CUDA support for our companion library targeting GPGPUs (HPXCL[3490]).
- 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)[3491]) who contributed to *HPX* the implementation of several parallel algorithms (as proposed by N4313[3492]).
- Anuj Sharma and Christopher Bross (Department of Computer Science 3 - Computer Architecture[3493]), who worked on *HPX* in the context of the Google Summer of Code[3494] program 2014.
- Martin Stumpf (Department of Computer Science 3 - Computer Architecture[3495]), who rebuilt our contiguous testing infrastructure (see the *HPX* Buildbot Website[3496]). Martin is also working on HPXCL[3497] (mainly all

[3483] https://www.lsu.edu
[3484] https://www.libgeodecomp.org/
[3485] https://www.libgeodecomp.org/
[3486] https://www.lsu.edu
[3487] https://www.conan.io/
[3488] https://www.lsu.edu
[3489] https://uoregon.edu/
[3490] https://github.com/STEllAR-GROUP/hpxcl/
[3491] https://www.cct.lsu.edu
[3492] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
[3493] https://www3.cs.fau.de
[3494] https://developers.google.com/open-source/soc/
[3495] https://www3.cs.fau.de
[3496] https://rostam.cct.lsu.edu/
[3497] https://github.com/STEllAR-GROUP/hpxcl/

work related to OpenCL[3498]) and implementing an *HPX* backend for POCL[3499], a portable computing language solution based on OpenCL[3500].

- Grant Mercer (University of Nevada, Las Vegas[3501]), who helped creating many of the parallel algorithms (as proposed by N4313[3502]).
- Damond Howard (Louisiana State University (LSU)[3503]), who works on HPXCL[3504] (mainly all work related to CUDA[3505]).
- 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[3506]. He also contributed an implementation of an API similar to Fortran co-arrays on top of *HPX*.
- John Biddiscombe (Swiss National Supercomputing Centre[3507]), 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.
- Patricia Grubel (New Mexico State University[3508]), 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*.

[3498] https://www.khronos.org/opencl/
[3499] https://portablecl.org/
[3500] https://www.khronos.org/opencl/
[3501] https://www.unlv.edu
[3502] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
[3503] https://www.lsu.edu
[3504] https://github.com/STEllAR-GROUP/hpxcl/
[3505] https://www.nvidia.com/object/cuda_home_new.html
[3506] https://www.numscale.com/nt2/
[3507] https://www.cscs.ch
[3508] https://www.nmsu.edu

- Joseph Kleinhenz, Markus Elfring, Kirill Kropivyansky, Alexander Neundorf, Bryant Lam, and Alex Hirsch who improved our CMake.
- Praveen Velliengiri, Jean-Loup Tastet, Michael Levine, Aalekh Nigam, HadrienG2, Prayag Verma, 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, and Matthew Anderson 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)[3509], Department of Computer Science 3 - Computer Architecture[3510], and Swiss National Supercomputing Centre[3511] 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 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).

[3509] https://www.cct.lsu.edu
[3510] https://www3.cs.fau.de
[3511] https://www.cscs.ch

CHAPTER 3

Index

- genindex

Index

Symbols

A

B

C

D

H

L

M

O

P

R

S