
HPX Documentation

master

The STE || AR Group

April 15, 2019

1	What is <i>HPX</i>?	3
2	What's so special about <i>HPX</i>?	5
2.1	Why <i>HPX</i> ?	5
2.2	Quick start	11
2.3	Terminology	17
2.4	Examples	18
2.5	Manual	40
2.6	Additional material	206
2.7	HPX Modules	206
2.8	API reference	207
2.9	Contributing to <i>HPX</i>	542
2.10	Releases	548
2.11	About <i>HPX</i>	704
3	Index	713
	Index	715

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

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

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

¹ <https://github.com/STELLAR-GROUP/hpx/issues>

² <https://cpplang.slack.com>

³ hpx-users@stellar.cct.lsu.edu

⁴ <https://stackoverflow.com/questions/tagged/hpx>

CHAPTER 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 STELLAR group (see [People](#)).

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

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

What's so special about *HPX*?

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

2.1 Why *HPX*?

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

2.1.1 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 Objects (LCOs)*
- Synchronization for anonymous producer-consumer scenarios
- Percolation (pre-staging of task data)

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

2.1.2 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 STEllAR⁵ Group (Systems Technology, Emergent Parallelism, and Algorithm Research) at Louisiana State University (LSU)⁶'s Center for Computation and Technology (CCT)⁷ to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STEllAR⁸ is a research group focusing on system software solutions and scientific application development for hybrid and many-core hardware architectures.

⁵ <https://stellar-group.org>

⁶ <https://www.lsu.edu>

⁷ <https://www.cct.lsu.edu>

⁸ <https://stellar-group.org>

For more information about the [STELLAR⁹](#) 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 [FLOPS¹⁰](#)—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 Law¹¹](#)) and weak scaling (see [Gustafson's Law¹²](#)). 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*:

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

Each of those four factors manifests itself in multiple and different ways; each of the hardware architectures and programming models expose specific forms. However the interesting part is that all of them are limiting the scalability of applications no matter what part of the hardware jungle we look at. Hand-helds, PCs, supercomputers, or the cloud, all suffer from the reign of the 4 horsemen: **Starvation**, **Latency**, **Overhead**, and **Contention**. This realization is very important as it allows us to derive the criteria for solutions to the scalability problem from first principles, 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 Neumann¹³](#), as published back in 1945, and later extended by the [Harvard architecture¹⁴](#). 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:

⁹ <https://stellar-group.org>

¹⁰ <http://en.wikipedia.org/wiki/FLOPS>

¹¹ http://en.wikipedia.org/wiki/Amdahl%27s_law

¹² http://en.wikipedia.org/wiki/Gustafson%27s_law

¹³ <http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf>

¹⁴ http://en.wikipedia.org/wiki/Harvard_architecture

- Performance: as mentioned, scalability and efficiency are the main criteria people are interested in
- Fault tolerance: the low expected mean time between failures (MTBF¹⁵) 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 turns 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*¹⁶, caching memory hierarchies in all modern processors, the constant optimization of existing *MPI*¹⁷ implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use *GPGPUs*¹⁸ today. It is important to note, that existing latencies are often tightly related to some resource having to wait for the operation to be completed. At the same time it would be perfectly fine to do some other, unrelated work in the meantime, allowing the system to hide the latencies by filling the idle-time with useful work. Modern systems already employ similar techniques (pipelined instruction execution in the processor cores, asynchronous input/output operations, and many more). What we propose is to go beyond anything we know today and to make latency hiding an intrinsic concept of the operation of the whole system stack.

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

¹⁵ http://en.wikipedia.org/wiki/Mean_time_between_failures

¹⁶ <http://en.wikipedia.org/wiki/InfiniBand>

¹⁷ https://en.wikipedia.org/wiki/Message_Passing_Interface

¹⁸ <http://en.wikipedia.org/wiki/GPGPU>

if we already have special machines supporting this mode of operation, such as the [Cray XMT](http://en.wikipedia.org/wiki/Cray_XMT)¹⁹). For conventional systems however, the smaller the overhead of a context switch and the finer the granularity of the threading system, the better will be the overall system utilization and its efficiency. For today's architectures we already see a flurry of libraries providing exactly this type of functionality: non-pre-emptive, task-queue based parallelization solutions, such as [Intel Threading Building Blocks \(TBB\)](https://www.threadingbuildingblocks.org/)²⁰, [Microsoft Parallel Patterns Library \(PPL\)](https://msdn.microsoft.com/en-us/library/dd492418.aspx)²¹, [Cilk++](https://software.intel.com/en-us/articles/intel-cilk-plus/)²², and many others. The possibility to suspend a current task if some preconditions for its execution are not met (such as waiting for I/O or the result of a different task), seamlessly switching to any other task which can continue, and to reschedule the initial task after the required result has been calculated, which makes the implementation of latency hiding almost trivial.

2.1.8 Rediscover constraint-based synchronization to replace global Barriers

The code we write today is riddled with implicit (and explicit) global barriers. By global barrier we mean the synchronization of the control flow between several (very often all) threads (when using [OpenMP](https://openmp.org/wp/)²³) or processes ([MPI](https://en.wikipedia.org/wiki/Message_Passing_Interface)²⁴). For instance, an implicit global barrier is inserted after each loop parallelized using [OpenMP](https://openmp.org/wp/)²⁵ as the system synchronizes the threads used to execute the different iterations in parallel. In [MPI](https://en.wikipedia.org/wiki/Message_Passing_Interface)²⁶ each of the communication steps imposes an explicit barrier onto the execution flow as (often all) nodes have to be synchronized. Each of those barriers 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](https://en.wikipedia.org/wiki/Message_Passing_Interface)²⁷ 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](https://www.pgas.org/)²⁸ (Partitioned Global Address Space) designed to overcome this limitation, such as [Chapel](https://chapel.cray.com/)²⁹, [X10](https://x10-lang.org/)³⁰, [UPC](https://upc.lbl.gov/)³¹, or [Fortress](https://labs.oracle.com/projects/plrg/Publications/index.html)³². However all systems based on [PGAS](https://www.pgas.org/)³³ rely

¹⁹ http://en.wikipedia.org/wiki/Cray_XMT

²⁰ <https://www.threadingbuildingblocks.org/>

²¹ <https://msdn.microsoft.com/en-us/library/dd492418.aspx>

²² <https://software.intel.com/en-us/articles/intel-cilk-plus/>

²³ <https://openmp.org/wp/>

²⁴ https://en.wikipedia.org/wiki/Message_Passing_Interface

²⁵ <https://openmp.org/wp/>

²⁶ https://en.wikipedia.org/wiki/Message_Passing_Interface

²⁷ https://en.wikipedia.org/wiki/Message_Passing_Interface

²⁸ <https://www.pgas.org/>

²⁹ <https://chapel.cray.com/>

³⁰ <https://x10-lang.org/>

³¹ <https://upc.lbl.gov/>

³² <https://labs.oracle.com/projects/plrg/Publications/index.html>

³³ <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++](https://charm.cs.uiuc.edu/)³⁴. The first attempts towards solving related problem go back decades as well, a good example is the [Linda coordination language](http://en.wikipedia.org/wiki/Linda_(coordination_language))³⁵. 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](http://en.wikipedia.org/wiki/GPGPU)³⁶ 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](https://en.wikipedia.org/wiki/Message_Passing_Interface)³⁷. This programming model is all about data transfer between nodes. [MPI](https://en.wikipedia.org/wiki/Message_Passing_Interface)³⁸ 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](https://en.wikipedia.org/wiki/Message_Passing_Interface)³⁹, 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++](https://charm.cs.uiuc.edu/)⁴⁰), 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](https://en.wikipedia.org/wiki/Message_Passing_Interface)⁴¹. 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](https://en.wikipedia.org/wiki/Message_Passing_Interface)⁴² 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++](https://charm.cs.uiuc.edu/)⁴³ 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

³⁴ <https://charm.cs.uiuc.edu/>

³⁵ [http://en.wikipedia.org/wiki/Linda_\(coordination_language\)](http://en.wikipedia.org/wiki/Linda_(coordination_language))

³⁶ <http://en.wikipedia.org/wiki/GPGPU>

³⁷ https://en.wikipedia.org/wiki/Message_Passing_Interface

³⁸ https://en.wikipedia.org/wiki/Message_Passing_Interface

³⁹ https://en.wikipedia.org/wiki/Message_Passing_Interface

⁴⁰ <https://charm.cs.uiuc.edu/>

⁴¹ https://en.wikipedia.org/wiki/Message_Passing_Interface

⁴² https://en.wikipedia.org/wiki/Message_Passing_Interface

⁴³ <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](https://stellar-group.org/downloads/)⁴⁴ 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](https://www.boost.org/)⁴⁵ and [Portable Hardware Locality \(HWLOC\)](https://www.open-mpi.org/projects/hwloc/)⁴⁶. 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](https://www.boost.org/)⁴⁷ and [Portable Hardware Locality \(HWLOC\)](https://www.open-mpi.org/projects/hwloc/)⁴⁸, it is recommended that you don't use the system allocator, but instead use either `tcmalloc` from [google-perftools](https://code.google.com/p/gperftools)⁴⁹ (default) or `jemalloc`⁵⁰ 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
```

⁴⁴ <https://stellar-group.org/downloads/>

⁴⁵ <https://www.boost.org/>

⁴⁶ <https://www.open-mpi.org/projects/hwloc/>

⁴⁷ <https://www.boost.org/>

⁴⁸ <https://www.open-mpi.org/projects/hwloc/>

⁴⁹ <https://code.google.com/p/gperftools>

⁵⁰ <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](https://www.cmake.org)⁵¹ 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](https://www.cmake.org)⁵² where to find them by passing one or more of the following options to [CMake](https://www.cmake.org)⁵³:

```
-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](https://www.cmake.org)⁵⁴.

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

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

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

```
make hello_world_1
```

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

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

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

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

⁵¹ <https://www.cmake.org>

⁵² <https://www.cmake.org>

⁵³ <https://www.cmake.org>

⁵⁴ <https://www.cmake.org>


```
... examples.quickstart.hello_world_2
... hello_world_2
... examples.quickstart.hello_world_1
... hello_world_1
... examples.quickstart.hello_world_distributed
... hello_world_distributed
```

It is also possible to build 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`⁵⁵ 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
```

⁵⁵ <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.

//[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;
}

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

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

(continues on next page)

(continued from previous page)

```

// 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), &rand_wrapper);

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

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

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

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

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](#)⁵⁶ 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

⁵⁶ <https://www.pgas.org/>

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](#)⁵⁷, the *J-machine*’s⁵⁸ support for remote method instantiation, and at the coarse grained variations of Unix remote procedure calls, among others. This enables work to be moved to the data as well as performing the more common action of bringing data to the work. A parcel can cause actions to occur remotely and asynchronously, among which are the creation of threads at different system nodes or synchronous domains.

Local Control Object

Lightweight Control Object

LCO A local control object (sometimes called a lightweight control object) is a general term for the synchronization mechanisms used in *HPX*. Any object implementing a certain concept can be seen as an LCO. This concept 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

⁵⁷ http://en.wikipedia.org/wiki/Dataflow_architecture

⁵⁸ <http://en.wikipedia.org/wiki/J%E2%80%93Machine>



Fig. 2.1: Schematic of a future execution.

is needed, a read operation is performed on the future. If the future's function hasn't completed when a read operation is performed on it, the reader *HPX*-thread is suspended until the future is ready. The future facility allows *HPX* to schedule work early in a program so that when the function value is needed it will already be calculated and available. We use this property in our Fibonacci example below to enable its parallel execution.

Setup

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

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

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

To run the program type:

```
./bin/fibonacci_local
```

This should print (time should be approximate):

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

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

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

Will yield:

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

Walkthrough

Now that you have compiled and run the code, let's look at how the code works. Since this code is written in C++, we will begin with the `main()` function. Here you can see that in *HPX*, `main()` is only used to initialize the runtime system. It is important to note that application-specific command line options are defined here. *HPX* uses [Boost.Program Options](#)⁵⁹ 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](#)⁶⁰ 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;
```

(continues on next page)

⁵⁹ https://www.boost.org/doc/html/program_options.html

⁶⁰ <https://www.boost.org/doc/>

(continued from previous page)

```

std::uint64_t r = fibonacci(n);

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

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

```

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

```

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

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

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

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

```

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

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

```

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.

```

(continues on next page)

(continued from previous page)

```

// However, we intentionally demonstrate it this way to create some
// heavy workload.

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

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

```

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

2.4.3 Remote execution with actions: Hello world

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

```

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

```

Setup

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

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

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

To run the program type:

```
./bin/hello_world_distributed
```

This should print:

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

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

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

will yield:

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

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

Walkthrough

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

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

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

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

    // The non-callback version of hpx::lcos::wait_all takes a single parameter,
    // 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 func-

tion 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`.

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
```

(continues on next page)

(continued from previous page)

```
> 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;

```

(continues on next page)

(continued from previous page)

```

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;
```

(continues on next page)

(continued from previous page)

```

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.



Fig. 2.2: Heat diffusion example program flow.

We parallelize this code over the following eight examples:

- Example 1
- Example 2
- Example 3

- Example 4
- Example 5
- Example 6
- Example 7
- Example 8

The first example is straight serial code. In this code we instantiate a vector `U` 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
    }
};
```

(continues on next page)

(continued from previous page)

```

        // 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
```

(continues on next page)

(continued from previous page)

```

        // 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 Downloads⁶¹](#) 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 Downloads⁶²](#).

2.5.2 *HPX* build system

The build system for *HPX* is based on [CMake⁶³](#). 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

[CMake⁶⁴](#) 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 [CMake⁶⁵](#) scripts try to adhere to the general cmake policies on how to write [CMake⁶⁶](#) 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.

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:

⁶¹ <https://stellar-group.org/downloads/>

⁶² <https://stellar-group.org/downloads/>

⁶³ <https://www.cmake.org>

⁶⁴ <https://www.cmake.org>

⁶⁵ <https://www.cmake.org>

⁶⁶ <https://www.cmake.org>

```
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](#)⁶⁷ interface.

1. Download and install CMake here: [CMake Downloads](#)⁶⁸. 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 your 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](#)⁶⁹ for a detailed overview of specific [CMake](#)⁷⁰ variables.

Options and variables

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

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

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

⁶⁷ <https://www.cmake.org>

⁶⁸ <https://www.cmake.org/cmake/resources/software.html>

⁶⁹ <https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/Useful-Variables#Compilers-and-Tools>

⁷⁰ <https://www.cmake.org>

```
cmake -UARIABLE 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 -DVARIBLE: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](https://www.boost.org/)⁷¹ and [Portable Hardware Locality \(HWLOC\)](https://www.open-mpi.org/projects/hwloc/)⁷². So, before you read further, please make sure you have a recent version of [Boost](https://www.boost.org/)⁷³ 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](#).

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](#)⁷⁴.

⁷¹ <https://www.boost.org/>

⁷² <https://www.open-mpi.org/projects/hwloc/>

⁷³ <https://www.boost.org/>

⁷⁴ <https://rostan.cct.lsu.edu/>

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

Name	Recommended version	Minimum version	Notes
Compilers			
GNU Compiler Collection (g++) ⁷⁵	4.9 or newer	4.9	
Intel Composer XE Suites ⁷⁶	2014 or newer	2014	
clang: a C language family frontend for LLVM ⁷⁷	3.8 or newer	3.8	
Build System			
CMake ⁷⁸	3.9.0	3.3.2	Cuda support 3.9
Required Libraries			
Boost C++ Libraries ⁷⁹	1.62.0 or newer	1.58.0	
Portable Hardware Locality (HWLOC) ⁸⁰	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 CMake⁸¹.

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.

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

Name	Recommended version	Minimum version	Notes
Compilers			
Visual C++ ⁸² (x64)	2015	2015	
Build System			
CMake ⁸³	3.9.0	3.3.2	
Required Libraries			
Boost ⁸⁴	1.62.0 or newer	1.58.0	
Portable Hardware Locality (HWLOC) ⁸⁵	1.11	1.5	

⁷⁵ <https://gcc.gnu.org>

⁷⁶ <https://software.intel.com/en-us/intel-composer-xe/>

⁷⁷ <https://clang.llvm.org/>

⁷⁸ <https://www.cmake.org>

⁷⁹ <https://www.boost.org/>

⁸⁰ <https://www.open-mpi.org/projects/hwloc/>

⁸¹ <https://www.cmake.org>

⁸² <https://msdn.microsoft.com/en-us/visualc/default.aspx>

⁸³ <https://www.cmake.org>

⁸⁴ <https://www.boost.org/>

⁸⁵ <https://www.open-mpi.org/projects/hwloc/>

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 parcellport, we currently recommend configuring *HPX* to use MPI so that MPI can be used for communication between different localities. Please set the CMake variable `MP I _ C X X _ C O M P I L E R` 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 ⁸⁶	1.7.1	1.7.1	Used as a replacement for the system allocator, and for allocation diagnostics.
libunwind ⁸⁷	0.99	0.97	Dependency of google-perftools on x86-64, used for stack unwinding.
Open MPI ⁸⁸	1.10.1	1.8.0	Can be used as a highspeed communication library backend for the parcellport.

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

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

Name	Recommended version	Minimum version	Notes
Performance Application Programming Interface (PAPI)	Used for accessing hardware performance data.		
jemalloc ⁸⁹	2.1.2	2.1.0	Used as a replacement for the system allocator.
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.

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

Name	Recommended version	Minimum version	Notes
Hierarchical Data Format V5 (HDF5) ⁹¹	1.8.7	1.6.7	Used for data I/O in some example applications. See important note below.

⁸⁶ <https://code.google.com/p/gperftools>

⁸⁷ <https://www.nongnu.org/libunwind>

⁸⁸ <https://www.open-mpi.org>

⁸⁹ <https://www.canonware.com/jemalloc>

⁹⁰ <https://www.hdfgroup.org/HDF5>

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](#)⁹² dependencies are not available through your system package manager you can install them using the [Python](#)⁹³ package manager `pip`:

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

You may need to set the following [CMake](#)⁹⁴ variables to make sure [CMake](#)⁹⁵ can find the required dependencies.

DOXYGEN_ROOT:PATH

Specifies where to look for the installation of the [Doxygen](#)⁹⁶ tool.

SPHINX_ROOT:PATH

Specifies where to look for the installation of the [Sphinx](#)⁹⁷ tool.

BREATHE_APIDOC_ROOT:PATH

Specifies where to look for the installation of the [Breathe](#)⁹⁸ tool.

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, pass-

⁹¹ <https://www.hdfgroup.org/HDF5>

⁹² <https://www.python.org>

⁹³ <https://www.python.org>

⁹⁴ <https://www.cmake.org>

⁹⁵ <https://www.cmake.org>

⁹⁶ <https://www.doxygen.org>

⁹⁷ <http://www.sphinx-doc.org>

⁹⁸ <https://breathe.readthedocs.io/en/latest>

ing `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](#)⁹⁹. Unpack the downloaded archive into a directory of your choosing. We will refer to this directory as `$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](#)¹⁰⁰. Unpack the downloaded archive into a directory of your choosing. We will refer to this directory as `$HWLOC`.

To build `hwloc` run:

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

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

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

Please see [Hwloc Documentation](#)¹⁰¹ for more information about `Hwloc`.

⁹⁹ <https://www.boost.org/users/download/>

¹⁰⁰ <https://www.open-mpi.org/software/hwloc/v1.11>

¹⁰¹ <https://www.open-mpi.org/projects/hwloc/doc/>

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.
 - `partitioned_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.

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 CMake¹⁰² 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 CMake¹⁰³ variables:

¹⁰² <https://www.cmake.org>

¹⁰³ <https://www.cmake.org>

- `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](#)¹⁰⁴). 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](#)¹⁰⁵ builds, this means that the `CMAKE_BUILD_TYPE` variables have to match and for projects not using [CMake](#)¹⁰⁶, 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](#)¹⁰⁷'s support for different toolchains (see [cmake-toolchains\(7\)](#)¹⁰⁸ for more information). This is also used for cross compilation.

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

In order to enable full static linking with the libraries, the [CMake](#)¹⁰⁹ variable `HPX_WITH_STATIC_LINKING:BOOL` has to be set to `On`.

Debugging applications using core files

For *HPX* to generate useful core files, *HPX* has to be compiled without signal and exception handlers `HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL`. If this option is not specified, the signal handlers change the application state. For example, after a segmentation fault the stack trace will show the signal handler. Similarly, unhandled exceptions are also caught by these handlers and the stack trace will not point to the location where the unhandled exception was thrown.

¹⁰⁴ https://cmake.org/cmake/help/latest/variable/CMAKE_BUILD_TYPE.html

¹⁰⁵ <https://www.cmake.org>

¹⁰⁶ <https://www.cmake.org>

¹⁰⁷ <https://www.cmake.org>

¹⁰⁸ <https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html>

¹⁰⁹ <https://www.cmake.org>

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:

```
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
```

(continues on next page)

(continued from previous page)

```
./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
```

- 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.
- If you want to use *HPX* with MPI, you need to enable the MPI parcellport, 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](https://www.boost.org/users/download/)¹¹⁰
- Install the boost library as explained in the section *Installing Boost*
- Install the hwloc library as explained in the section *Installing Hwloc*

¹¹⁰ <https://www.boost.org/users/download/>

- Download the latest version of CMake binaries, which are located under the platform section of the downloads page at [CMake Downloads](#)¹¹¹.
- Download the latest version of HPX from the STEllAR website: [HPX Downloads](#)¹¹².

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](#)¹¹³ and [Portable Hardware Locality \(HWLOC\)](#)¹¹⁴ 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.
- Build the INSTALL target.

For more detailed information about using [CMake](#)¹¹⁵ 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](#)¹¹⁶ V3.4.3 installer (or latest version) from [here](#)¹¹⁷
- Download the [Portable Hardware Locality \(HWLOC\)](#)¹¹⁸ V1.11.0 (or latest version) from [here](#)¹¹⁹ and unpack it.

¹¹¹ <https://www.cmake.org/cmake/resources/software.html>

¹¹² <https://stellar-group.org/downloads/>

¹¹³ <https://www.boost.org/>

¹¹⁴ <https://www.open-mpi.org/projects/hwloc/>

¹¹⁵ <https://www.cmake.org>

¹¹⁶ <https://www.cmake.org>

¹¹⁷ <https://blog.kitware.com/cmake-3-4-3-available-for-download/>

¹¹⁸ <https://www.open-mpi.org/projects/hwloc/>

¹¹⁹ <http://www.open-mpi.org/software/hwloc/v1.11/downloads/hwloc-win64-build-1.11.0.zip>

- Download [Boost](https://www.boost.org/)¹²⁰ libraries V1.60 (or latest version) from [here](https://sourceforge.net/projects/boost/files/boost/1.60.0/)¹²¹ 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:



Fig. 2.3: Example CMake path settings.

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.

¹²⁰ <https://www.boost.org/>

¹²¹ <https://sourceforge.net/projects/boost/files/boost/1.60.0/>

- Set three new environment variables (in CMake, not in Windows environment, by the way): `BOOST_ROOT`, `HWLOC_ROOT`, `CMAKE_INSTALL_PREFIX`. 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.

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:



Fig. 2.5: Visual Studio INSTALL target.

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

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](https://trac.alcf.anl.gov/projects/llvm-bgq)¹²².
- Build (and install) a recent version of [Hwloc Downloads](https://www.open-mpi.org/software/hwloc/v1.11)¹²³. 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
```

¹²² <https://trac.alcf.anl.gov/projects/llvm-bgq>

¹²³ <https://www.open-mpi.org/software/hwloc/v1.11>



```

Output
Show output from: Build
116> -- Installing: C:/bin/HPX/bin/1d_stencil_2.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_3.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_4.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_4_parallel.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_5.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_6.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_7.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_8.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_1_omp.exe
116> -- Installing: C:/bin/HPX/bin/1d_stencil_3_omp.exe
116> -- Installing: C:/bin/HPX/bin/simple_central_tuplespace_client.exe
116> -- Installing: C:/bin/HPX/lib/hpx_simple_central_tuplespaced.lib
116> -- Installing: C:/bin/HPX/lib/hpx_simple_central_tuplespaced.dll
116> -- Installing: C:/bin/HPX/bin/transpose_serial.exe
116> -- Installing: C:/bin/HPX/bin/transpose_serial_block.exe
116> -- Installing: C:/bin/HPX/bin/transpose_smp.exe
116> -- Installing: C:/bin/HPX/bin/transpose_smp_block.exe
116> -- Installing: C:/bin/HPX/bin/transpose_block.exe
116> -- Installing: C:/bin/HPX/bin/transpose_serial_vector.exe
116> -- Installing: C:/bin/HPX/bin/hpx_runtime.exe
===== Build: 116 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
Error List Output Find Symbol Results Package Manager Console Azure App Service Activity

```

Fig. 2.6: Visual Studio build output.

- 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 \

```

(continues on next page)

(continued from previous page)

```
-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:

```
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](#)¹²⁴ 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](#)¹²⁵, unpack the retrieved tarball and change to the newly created directory.
- Run the configure-make-install procedure as follows:

¹²⁴ <https://www.boost.org/users/download/>

¹²⁵ <https://www.open-mpi.org/software/hwloc/v1.11>

```
CC=icc CFLAGS=-mmic CXX=icpc CXXFLAGS=-mmic LDFLAGS=-mmic ./configure --host=x86_
↳ 64-knom-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.

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*¹²⁶ page.

Note: This section of the manual is based off of our collaborators Patrick Diehl's blog post *Installing HPX on Fedora 22*¹²⁷.

- 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:

¹²⁶ <https://stellar-group.org/downloads/>

¹²⁷ <http://diehlpk.github.io/2015/08/04/hpx-fedora.html>

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

Note: To build HPX without examples use:

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

- Add the library path of HPX to ldconfig:

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

How to install *HPX* on Arch distributions

Important: There are *HPX* packages for Arch in the AUR. Unless you want to customize your build you may want to start off with those. Instructions can be found on the [HPX Downloads](https://stellar-group.org/downloads)¹²⁸ 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
```

¹²⁸ <https://stellar-group.org/downloads/>

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*
- *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-gnueabi-g++-4.8)
set(CMAKE_C_COMPILER arm-linux-gnueabi-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
#
```

(continues on next page)

(continued from previous page)

```

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_PARCELPORIBVERBS ON CACHE BOOL "")
# Always disable the tcp parcellport as it is non-functional on the BGQ.
set(HPX_WITH_PARCELPORITCP ON CACHE BOOL "")
# Always enable the tcp parcellport as it is currently the only way to communicate on
↪the BGQ.
set(HPX_WITH_PARCELPORIMPI 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 @ bbpb2.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")

```

(continues on next page)

(continued from previous page)

```
# Always disable the ibverbs parcellport as it is non-functional on the BGQ.
set(HPX_WITH_IBVERBS_PARCELPORT OFF)
# Always disable the tcp parcellport as it is non-functional on the BGQ.
set(HPX_WITH_TCP_PARCELPORT OFF)
# Always enable the tcp parcellport 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 "")
```

(continues on next page)

(continued from previous page)

```

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_PARCELPOR TCP ON CACHE BOOL "")
set(HPX_WITH_PARCELPOR MPI ON CACHE BOOL "")
set(HPX_WITH_PARCELPOR MPI_MULTITHREADED OFF CACHE BOOL "")
set(HPX_WITH_PARCELPOR LIBFABRIC ON CACHE BOOL "")
set(HPX_PARCELPOR_LIBFABRIC_PROVIDER "gni" CACHE STRING
  "See libfabric docs for details, gni, verbs, psm2 etc etc")
set(HPX_PARCELPOR_LIBFABRIC_THROTTLE_SENDS "256" CACHE STRING
  "Max number of messages in flight at once")
set(HPX_PARCELPOR_LIBFABRIC_WITH_DEV_MODE OFF CACHE BOOL
  "Custom libfabric logging flag")
set(HPX_PARCELPOR_LIBFABRIC_WITH_LOGGING OFF CACHE BOOL
  "Libfabric parcellport logging on/off flag")
set(HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD "4096" CACHE STRING
  "The threshold in bytes to when perform zero copy optimizations (default: 128)")
# We do a cross compilation here ...
set(CMAKE_CROSSCOMPILING ON CACHE BOOL "")

```

CrayKNL

```

# Copyright (c) 2014 Thomas Heller
#
# 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)

```

(continues on next page)

(continued from previous page)

```

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_PARCELPOR TCP ON CACHE BOOL "")
set(HPX_WITH_PARCELPOR MPI ON CACHE BOOL "")
set(HPX_WITH_PARCELPOR MPI_MULTITHREADED OFF CACHE BOOL "")
set(HPX_WITH_PARCELPOR LIBFABRIC ON CACHE BOOL "")
set(HPX_PARCELPOR_LIBFABRIC_PROVIDER "gni" CACHE STRING

```

(continues on next page)

(continued from previous page)

```

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

```

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)

```

(continues on next page)

(continued from previous page)

```

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

```

CrayStatic

```

# Copyright (c) 2014-2017 Thomas Heller
# Copyright (c) 2017 Bryce Adelstein Lelbach
#
# 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 "")

```

(continues on next page)

(continued from previous page)

```

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_PARCELPOR_TCP ON CACHE BOOL "")
set(HPX_WITH_PARCELPOR_MPI ON CACHE BOOL "")
set(HPX_WITH_PARCELPOR_MPI_MULTITHREADED ON CACHE BOOL "")
set(HPX_WITH_PARCELPOR_LIBFABRIC ON CACHE BOOL "")
set(HPX_PARCELPOR_LIBFABRIC_PROVIDER "gni" CACHE STRING
  "See libfabric docs for details, gni,verbs,psm2 etc etc")
set(HPX_PARCELPOR_LIBFABRIC_THROTTLE_SENDS "256" CACHE STRING
  "Max number of messages in flight at once")
set(HPX_PARCELPOR_LIBFABRIC_WITH_DEV_MODE OFF CACHE BOOL
  "Custom libfabric logging flag")
set(HPX_PARCELPOR_LIBFABRIC_WITH_LOGGING OFF CACHE BOOL
  "Libfabric parcellport logging on/off flag")
set(HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD "4096" CACHE STRING
  "The threshold in bytes to when perform zero copy optimizations (default: 128)")

```

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")

```

(continues on next page)

(continued from previous page)

```
# 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/Phi
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_DEPRECATED_WARNINGS:BOOL`
- `HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL`
- `HPX_WITH_DYNAMIC_HPX_MAIN:BOOL`
- `HPX_WITH_FAULT_TOLERANCE:BOOL`
- `HPX_WITH_FORTTRAN: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_DEPRECATED_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 threshold in bytes to when perform zero copy optimizations (default: 128)

Build Targets options

- `HPX_WITH_COMPILE_ONLY_TESTS:BOOL`
- `HPX_WITH_DEFAULT_TARGETS:BOOL`
- `HPX_WITH_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 than this number of OS-Threads (empty string means dynamic) (default: 64)

HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING

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

HPX_WITH_THREAD_TARGET_ADDRESS:BOOL

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

HPX_WITH_TIMER_POOL:BOOL

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

AGAS options

- *HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL*

HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL

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

Parcelport options

- *HPX_WITH_NETWORKING:BOOL*
- *HPX_WITH_PARCELPORT_ACTION_COUNTERS:BOOL*
- *HPX_WITH_PARCELPORT_LIBFABRIC:BOOL*
- *HPX_WITH_PARCELPORT_MPI:BOOL*
- *HPX_WITH_PARCELPORT_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_PARCELPART_ACTION_COUNTERS:BOOL

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

HPX_WITH_PARCELPART_LIBFABRIC:BOOL

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

HPX_WITH_PARCELPART_MPI:BOOL

Enable the MPI based parcellport.

HPX_WITH_PARCELPART_MPI_ENV:STRING

List of environment variables checked to detect MPI (default: MV2_COMM_WORLD_RANK;PMI_RANK;OMPI_COMM_WO

HPX_WITH_PARCELPART_MPI_MULTITHREADED:BOOL

Turn on MPI multithreading support (default: ON).

HPX_WITH_PARCELPART_TCP:BOOL

Enable the TCP based parcellport.

HPX_WITH_PARCELPART_VERBS:BOOL

Enable the ibverbs based parcellport. 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_SANITIZERS: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_SANITIZERS:BOOL

Configure with sanitizer instrumentation support.

HPX_WITH_TESTS_DEBUG_LOG:BOOL

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

HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING

Destination for test debug logs (default: `cout`)

HPX_WITH_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`¹²⁹ installation to be used for compiling *HPX*. Set this if CMake is not able to locate a suitable version of `Boost`¹³⁰. The directory specified here can be either the root of a installed Boost distribution or the directory where you unpacked and built `Boost`¹³¹ without installing it (with staged libraries).

¹²⁹ <https://www.boost.org/>

¹³⁰ <https://www.boost.org/>

¹³¹ <https://www.boost.org/>

HWLOC_ROOT:PATH

Specifies where to look for the [Portable Hardware Locality \(HWLOC\)](#)¹³² library. Set this if CMake is not able to locate a suitable version of [Portable Hardware Locality \(HWLOC\)](#)¹³³. [Portable Hardware Locality \(HWLOC\)](#)¹³⁴ 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 \(PAPI\)](#)¹³⁵ 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.

```
// 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.

//[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;
}
```

(continues on next page)

¹³² <https://www.open-mpi.org/projects/hwloc/>

¹³³ <https://www.open-mpi.org/projects/hwloc/>

¹³⁴ <https://www.open-mpi.org/projects/hwloc/>

¹³⁵ <https://icl.cs.utk.edu/papi/>

(continued from previous page)

```
}
//]
```

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;
    }
}}
```

(continues on next page)

(continued from previous page)

```

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();

```

(continues on next page)

(continued from previous page)

```

    }
};
}

#endif // HELLO_WORLD_COMPONENT_HPP

```

hello_world_client.cpp

```

// Copyright (c) 2012 Bryce Lebach
//
// 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`¹³⁶ 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
- 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:

¹³⁶ https://www.cmake.org/cmake/help/latest/command/find_package.html

- 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`¹³⁷ 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

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 `RPATHs`, you can include `HPX_SetFullRPATH` in your project after all libraries you want to link against have been added. Please also consult the CMake documentation [here](https://www.cmake.org/cmake/help/latest/command/target_link_libraries.html)¹³⁸.

¹³⁷ https://www.cmake.org/cmake/help/latest/command/target_link_libraries.html

¹³⁸ <https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/RPATH-handling>

Using HPX with Makefile

A basic project building with *HPX* is through creating makefiles. The process of creating one can get complex depending upon the use of cmake parameter `HPX_WITH_HP_X_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.
//
// [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
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_HP_X_MAIN=OFF
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```

#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 Lebach
//
// 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>

```

(continues on next page)

(continued from previous page)

```

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)

```

(continues on next page)

(continued from previous page)

To build the program, type:

```
make
```

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

```
./hello_world
```

Note: Due to high variations in CMake flags and library dependencies, it is recommended to build *HPX* applications and components with `pkg-config` or `CMakeLists.txt`. Writing Makefile may result in broken builds if due care is not taken. `pkg-config` files and CMake systems are configured with CMake build of *HPX*. Hence, they are stable 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 than `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 as GUIs. The function `hpx::stop` can be used to wait for the *HPX* runtime system to exit and should be at least used as the last function called in `main()`. The value returned from `hpx_main` will be returned from `hpx::stop` after the runtime system has stopped.

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

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

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](#)¹³⁹. This is a structured text format allowing to group key/value pairs (properties) into sections. The basic element contained in an ini file is the property. Every property has a name and a value, delimited by an equals sign '='. The name appears to the left of the equals sign:

```
name=value
```

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

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

```
[section]
```

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

```
[outer_section.inner_section]
```

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

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

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

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

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

If a property with the same name is reused inside a section, the second occurrence of this property name will override the first occurrence (discard the first value). Duplicate sections simply merge their properties together, as if they occurred contiguously.

In *HPX* ini files, a property value `${FOO:default}` will use the environmental variable `FOO` to extract the actual value if it is set and `default` otherwise. No default has to be specified. Therefore `${FOO}` refers to the environmental variable `FOO`. If `FOO` is not set or empty the overall expression will evaluate to an empty string. A property value `[$section.key:default]` refers to the value held by the property `section.key` if it exists and `default`

¹³⁹ https://en.wikipedia.org/wiki/INI_file

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
<code>system.pid</code>	This is initialized to store the current OS-process id of the application instance.
<code>system.prefix</code>	This is initialized to the base directory <i>HPX</i> has been loaded from.
<code>system.executable_prefix</code>	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}
```

(continues on next page)

(continued from previous page)

```
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
<code>hpx.location</code>	This is initialized to the id of the <i>locality</i> this application instance is running on.
<code>hpx.component</code>	Duplicates are discarded. This property can refer to a list of directories separated by ':' (Linux, Android, and MacOS) or using ';' (Windows).
<code>hpx.master_ini_path</code>	This is initialized to the list of default paths of the main <code>hpx.ini</code> configuration files. This property can refer to a list of directories separated by ':' (Linux, Android, and MacOS) or using ';' (Windows).
<code>hpx.ini_path</code>	This is initialized to the default path where <i>HPX</i> 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).
<code>hpx.os_threads</code>	This setting reflects the number of OS-threads used for running <i>HPX</i> -threads. Defaults to number of detected cores (not hyperthreads/PUs).
<code>hpx.localities</code>	This setting reflects the number of localities the application is running on. Defaults to 1.
<code>hpx.program_name</code>	This setting reflects the program name of the application instance. Initialized from the command line <code>argv[0]</code> .
<code>hpx.cmd_line</code>	This setting reflects the actual command line used to launch this application instance.
<code>hpx.lock_detection</code>	This setting verifies that no locks are being held while a <i>HPX</i> thread is suspended. This setting is applicable only if <code>HPX_WITH_VERIFY_LOCKS</code> is set during configuration in CMake.
<code>hpx.throw_on_held_lock</code>	This setting causes an exception if during lock detection at least one lock is being held while a <i>HPX</i> thread is suspended. This setting is applicable only if <code>HPX_WITH_VERIFY_LOCKS</code> is set during configuration in CMake. This setting has no effect if <code>hpx.lock_detection=0</code> .
<code>hpx.minimal_deadlock_detection</code>	This setting enables support for minimal deadlock detection for <i>HPX</i> -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 <code>HPX_WITH_THREAD_DEADLOCK_DETECTION</code> is set during configuration in CMake.
<code>hpx.spinlock_deadlock_detection_limit</code>	This setting verifies that spinlocks don't spin longer than specified using the <code>hpx.spinlock_deadlock_detection_limit</code> . This setting is applicable only if <code>HPX_WITH_SPINLOCK_DEADLOCK_DETECTION</code> is set during configuration in CMake. By default this is set to 1 (for Debug builds) or to 0 (for Release, RelWithDebInfo, RelMinSize builds).
<code>hpx.spinlock_deadlock_detection_upper_limit</code>	This setting specifies the upper limit of allowed number of spins that spinlocks are allowed to perform. This setting is applicable only if <code>HPX_WITH_SPINLOCK_DEADLOCK_DETECTION</code> is set during configuration in CMake. By default this is set to 1000000.
<code>hpx.max_background_threads</code>	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.
<code>hpx.max_idle_loop_count</code>	By default this is defined by the preprocessor constant <code>HPX_IDLE_LOOP_COUNT_MAX</code> . This is an internal setting which you should change only if you know exactly what you are doing.
<code>hpx.max_busy_loop_count</code>	This setting defines the maximum value of the busy-loop counter in the scheduler. By default this is defined by the preprocessor constant <code>HPX_BUSY_LOOP_COUNT_MAX</code> . This is an internal setting which you should change only if you know exactly what you are doing.
<code>hpx.max_idle_backoff_time</code>	This setting defines the maximum time (in milliseconds) for the scheduler to sleep after being idle for <code>hpx.max_idle_loop_count</code> iterations. This setting is applicable only if <code>HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF</code> is set during configuration in CMake. By default this is defined by the preprocessor constant <code>HPX_IDLE_BACKOFF_TIME_MAX</code> . This is an internal setting which you should change only if you know exactly what you are doing.
<code>hpx.stacks.small_size</code>	This is initialized to the small stack size to be used by <i>HPX</i> -threads. Set by default to the value of the compile time preprocessor constant <code>HPX_SMALL_STACK_SIZE</code> (defaults to 0x8000). This value is used for all <i>HPX</i> threads by default, except for the thread running <code>hpx_main</code> (which runs on a large stack).
<code>hpx.stacks.medium_size</code>	This is initialized to the medium stack size to be used by <i>HPX</i> -threads. Set by default to the value of the compile time preprocessor constant <code>HPX_MEDIUM_STACK_SIZE</code> (defaults to 0x20000).
<code>hpx.stacks.large_size</code>	This is initialized to the large stack size to be used by <i>HPX</i> -threads. Set by default to the value of the compile time preprocessor constant <code>HPX_LARGE_STACK_SIZE</code> (defaults to 0x200000). This setting is used by default for the thread running <code>hpx_main</code> only.
<code>hpx.stacks.huge_size</code>	This is initialized to the huge stack size to be used by <i>HPX</i> -threads. Set by default to the value of the compile time preprocessor constant <code>HPX_HUGE_STACK_SIZE</code> (defaults to 0x400000).

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
<code>hpx.threadpools.io_pool_size</code>	The value of this property defines the number of OS-threads created for the internal I/O thread pool.
<code>hpx.threadpools.parcel_pool_size</code>	The value of this property defines the number of OS-threads created for the internal parcel thread pool.
<code>hpx.threadpools.timer_pool_size</code>	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
<code>hpx.thread_queue.min_tasks_to_steal_pending</code>	The value of this property defines the number of pending <i>HPX</i> threads which have to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.
<code>hpx.thread_queue.min_tasks_to_steal_staged</code>	The value of this property defines the number of staged <i>HPX</i> 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.
<code>hpx.thread_queue.min_add_new_count</code>	The value of this property defines the minimal number tasks to be converted into <i>HPX</i> threads whenever the thread queues for a core have run empty.
<code>hpx.thread_queue.max_add_new_count</code>	The value of this property defines the maximal number tasks to be converted into <i>HPX</i> threads whenever the thread queues for a core have run empty.
<code>hpx.thread_queue.max_delete_count</code>	The value of this property defines the number number of terminated <i>HPX</i> 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
<code>hpx.components.load_external</code>	This entry defines whether external components will be loaded on this <i>locality</i> . 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 <i>AGAS</i> server <i>locality</i> .

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
<code>hpx.components.<component_instance_name></code>	This is the name of a component, usually the same as the second argument to the macro used while registering the component with <code>HPX_REGISTER_COMPONENT</code> . Set by the <code>component_factory</code> .
<code>hpx.components.<component_instance_name>.path</code>	This is either the full path file name of the component module or the directory the component module is located in. In this case, the component module name will be derived from the property <code>hpx.components.<component_instance_name>.name</code> . Set by the <code>component factory</code> .
<code>hpx.components.<component_instance_name>.enabled</code>	This setting explicitly enables or disables the component. This is an optional property, <i>HPX</i> 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 <i>parcel</i> layer to listen to. This IP address will be used as long as no other values are specified (for instance using the <code>--hpx:hpx</code> command line option). The expected format is any valid IP address or domain name format which can be resolved into an IP address. The default depends on the compile time preprocessor constant <code>HPX_INITIAL_IP_ADDRESS ("127.0.0.1")</code> .
hpx. parcel. port	This property defines the default IP port to be used for the <i>parcel</i> layer to listen to. This IP port will be used as long as no other values are specified (for instance using the <code>--hpx:hpx</code> command line option). The default depends on the compile time preprocessor constant <code>HPX_INITIAL_IP_PORT (7910)</code> .
hpx. parcel. bootstrap	This property defines which parcelport type should be used during application bootstrap. The default depends on the compile time preprocessor constant <code>HPX_PARCEL_BOOTSTRAP ("tcp")</code> .
hpx. parcel. max_connections	This property defines how many network connections between different localities are overall kept alive by each of <i>locality</i> . The default depends on the compile time preprocessor constant <code>HPX_PARCEL_MAX_CONNECTIONS (512)</code> .
hpx. parcel. max_connections_per_locality	This property defines the maximum number of network connections that one <i>locality</i> will open to another <i>locality</i> . The default depends on the compile time preprocessor constant <code>HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY (4)</code> .
hpx. parcel. max_message_size	This property defines the maximum allowed message size which will be transferrable through the <i>parcel</i> layer. The default depends on the compile time preprocessor constant <code>HPX_PARCEL_MAX_MESSAGE_SIZE (1000000000 bytes)</code> .
hpx. parcel. max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size which will be transferrable through the <i>parcel</i> layer. The default depends on the compile time preprocessor constant <code>HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE (1000000 bytes)</code> .
hpx. parcel. array_optimization	This property defines whether this <i>locality</i> is allowed to utilize array optimizations during serialization of <i>parcel</i> data. The default is 1.
hpx. parcel. zero_copy_optimization	This property defines whether this <i>locality</i> is allowed to utilize zero copy optimizations during serialization of <i>parcel</i> data. The default is the same value as set for <code>hpx.parcel.array_optimization</code> .
hpx. parcel. async_serialization	This property defines whether this <i>locality</i> 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 = ${HPX_HAVE_PARCELPOR_TCP:${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}}
```

(continues on next page)

(continued from previous page)

```

max_message_size = ${HPX_PARCEL_TCP_MAX_MESSAGE_SIZE:${hpx.parcels.max_message_size}}
max_outbound_message_size = ${HPX_PARCEL_TCP_MAX_OUTBOUND_MESSAGE_SIZE:${hpx.parcels.
↪max_outbound_message_size}}

```

Property	Description
<code>hpx.parcels.tcp.enable</code>	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).
<code>hpx.parcels.tcp.array_optimization</code>	This property defines whether this <i>locality</i> 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 <code>hpx.parcels.array_optimization</code> .
<code>hpx.parcels.tcp.zero_copy_optimization</code>	This property defines whether this <i>locality</i> 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 <code>hpx.parcels.zero_copy_optimization</code> .
<code>hpx.parcels.tcp.async_serialization</code>	This property defines whether this <i>locality</i> 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 <code>hpx.parcels.async_serialization</code> .
<code>hpx.parcels.tcp.parcels_pool_size</code>	The value of this property defines the number of OS-threads created for the internal parcel thread pool of the TCP <i>parcel</i> port. The default is taken from <code>hpx.threadpools.parcels_pool_size</code> .
<code>hpx.parcels.tcp.max_connections</code>	This property defines how many network connections between different localities are overall kept alive by each of <i>locality</i> . The default is taken from <code>hpx.parcels.max_connections</code> .
<code>hpx.parcels.tcp.max_connections_per_locality</code>	This property defines the maximum number of network connections that one <i>locality</i> will open to another <i>locality</i> . The default is taken from <code>hpx.parcels.max_connections_per_locality</code> .
<code>hpx.parcels.tcp.max_message_size</code>	This property defines the maximum allowed message size which will be transferable through the <i>parcel</i> layer. The default is taken from <code>hpx.parcels.max_message_size</code> .
<code>hpx.parcels.tcp.max_outbound_message_size</code>	This property defines the maximum allowed outbound coalesced message size which will be transferable through the <i>parcel</i> layer. The default is taken from <code>hpx.parcels.max_outbound_message_size</code> .

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

```

[hpx.parcels.mpi]
enable = ${HPX_HAVE_PARCELPORTRMPI:${hpx.parcels.enabled}}
env = ${HPX_HAVE_PARCELPORTRMPI_ENV:MV2_COMM_WORLD_RANK,PMI_RANK,OMPI_COMM_WORLD_SIZE,
↪ALPS_APP_PE}
multithreaded = ${HPX_HAVE_PARCELPORTRMPI_MULTITHREADED:0}
rank = <MPI_rank>
processor_name = <MPI_processor_name>
array_optimization = ${HPX_HAVE_PARCELPORTRMPI_ARRAY_OPTIMIZATION:${hpx.parcels.array_
↪optimization}}
zero_copy_optimization = ${HPX_HAVE_PARCELPORTRMPI_ZERO_COPY_OPTIMIZATION:${hpx.parcels.
↪zero_copy_optimization}}
use_io_pool = ${HPX_HAVE_PARCELPORTRMPI_USE_IO_POOL:$1}
async_serialization = ${HPX_HAVE_PARCELPORTRMPI_ASYNC_SERIALIZATION:${hpx.parcels.async_
↪serialization}}
parcels_pool_size = ${HPX_HAVE_PARCELPORTRMPI_PARCELPORTRMPI_POOL_SIZE:${hpx.threadpools.parcels_
↪pool_size}}

```

(continues on next page)

(continued from previous page)

```

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

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

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}
```

(continues on next page)

(continued from previous page)

```
[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
<code>hpx.commandline.aliases</code>	Enable command line aliases as defined in the section <code>hpx.commandline.aliases</code> (see below). Defaults to 1.
<code>hpx.commandline.allow_unknown</code>	Allow for unknown command line options to be passed through to <code>hpx_main()</code> . Defaults to 0.
<code>hpx.commandline.aliases.-a</code>	On the commandline, <code>-a</code> expands to: <code>--hpx:agas</code> .
<code>hpx.commandline.aliases.-c</code>	On the commandline, <code>-c</code> expands to: <code>--hpx:console</code> .
<code>hpx.commandline.aliases.-h</code>	On the commandline, <code>-h</code> expands to: <code>--hpx:help</code> .
<code>hpx.commandline.aliases.--help</code>	On the commandline, <code>--help</code> expands to: <code>--hpx:help</code> .
<code>hpx.commandline.aliases.-I</code>	On the commandline, <code>-I</code> expands to: <code>--hpx:ini</code> .
<code>hpx.commandline.aliases.-l</code>	On the commandline, <code>-l</code> expands to: <code>--hpx:localities</code> .
<code>hpx.commandline.aliases.-p</code>	On the commandline, <code>-p</code> expands to: <code>--hpx:app-config</code> .
<code>hpx.commandline.aliases.-q</code>	On the commandline, <code>-q</code> expands to: <code>--hpx:queuing</code> .
<code>hpx.commandline.aliases.-r</code>	On the commandline, <code>-r</code> expands to: <code>--hpx:run-agas-server</code> .
<code>hpx.commandline.aliases.-t</code>	On the commandline, <code>-t</code> expands to: <code>--hpx:threads</code> .
<code>hpx.commandline.aliases.-v</code>	On the commandline, <code>-v</code> expands to: <code>--hpx:version</code> .
<code>hpx.commandline.aliases.--version</code>	On the commandline, <code>--version</code> expands to: <code>--hpx:version</code> .
<code>hpx.commandline.aliases.-w</code>	On the commandline, <code>-w</code> expands to: <code>--hpx:worker</code> .
<code>hpx.commandline.aliases.-x</code>	On the commandline, <code>-x</code> expands to: <code>--hpx:hpx</code> .
<code>hpx.commandline.aliases.-0</code>	On the commandline, <code>-0</code> expands to: <code>--hpx:node=0</code> .
<code>hpx.commandline.aliases.-1</code>	On the commandline, <code>-1</code> expands to: <code>--hpx:node=1</code> .
<code>hpx.commandline.aliases.-2</code>	On the commandline, <code>-2</code> expands to: <code>--hpx:node=2</code> .
<code>hpx.commandline.aliases.-3</code>	On the commandline, <code>-3</code> expands to: <code>--hpx:node=3</code> .
<code>hpx.commandline.aliases.-4</code>	On the commandline, <code>-4</code> expands to: <code>--hpx:node=4</code> .
<code>hpx.commandline.aliases.-5</code>	On the commandline, <code>-5</code> expands to: <code>--hpx:node=5</code> .
<code>hpx.commandline.aliases.-6</code>	On the commandline, <code>-6</code> expands to: <code>--hpx:node=6</code> .
<code>hpx.commandline.aliases.-7</code>	On the commandline, <code>-7</code> expands to: <code>--hpx:node=7</code> .
<code>hpx.commandline.aliases.-8</code>	On the commandline, <code>-8</code> expands to: <code>--hpx:node=8</code> .
<code>hpx.commandline.aliases.-9</code>	On the commandline, <code>-9</code> expands to: <code>--hpx:node=9</code> .

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>
    {
    public:
        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.
```

(continues on next page)

(continued from previous page)

```

    hpx::naming::id_type app_server_instance =
        hpx::create_component<app::server>(hpx::find_here());

    // Create an instance of the action 'print_greeting_action'.
    app::server::print_greeting_action print_greeting;

    // Invoke the action 'print_greeting' on the newly created component.
    print_greeting(app_server_instance);
}
return hpx::finalize();
}

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

```

In order to make sure that the application will be able to use the component `app::server`, special configuration information must be passed to *HPX*. The simplest way to allow *HPX* to ‘find’ the component is to provide special ini configuration files, 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 <i>HPX</i> , such as thread-manager, parcel layer, LCOs, etc.	HPX_LOGLEVEL
<i>AGAS</i>	AGAS	Logging output generated by the <i>AGAS</i> 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>.log` (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
<code>file(<filename>)</code>	Direct all output to a file with the given <code><filename></code> .
<code>cout</code>	Direct all output to the local standard output of the application instance on this <i>locality</i> .
<code>cerr</code>	Direct all output to the local standard error output of the application instance on this <i>locality</i> .
<code>console</code>	Direct all output to the console instance of the application. The console instance has its logging destinations configured separately.
<code>android_log</code>	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
<i>locality</i>	The id of the <i>locality</i> on which the logging message was generated.
hpxthread	The id of the <i>HPX</i> -thread generating this logging output.
hpxphase	The phase ¹⁴⁰ of the <i>HPX</i> -thread generating this logging output.
hpxcomponent	The local virtual address of the component which the current <i>HPX</i> -thread is accessing.
parentloc	The id of the <i>locality</i> where the <i>HPX</i> thread was running which initiated the current <i>HPX</i> -thread. The current <i>HPX</i> -thread is generating this logging output.
hpxparent	The id of the <i>HPX</i> -thread which initiated the current <i>HPX</i> -thread. The current <i>HPX</i> -thread is generating this logging output.
hpxparentphase	The phase of the <i>HPX</i> -thread when it initiated the current <i>HPX</i> -thread. The current <i>HPX</i> -thread is generating this logging output.
time	The time stamp for this logging outputline as generated by the source <i>locality</i> .
idx	The sequence number of the logging output line as generated on the source <i>locality</i> .
osthread	The sequence number of the OS-thread which executes the current <i>HPX</i> -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 [0000000000000004d]
  <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.

¹⁴⁰ 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* parcellport 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	<code>--hpx:agas</code>
-c	<code>--hpx:console</code>
-h	<code>--hpx:help</code>
-I	<code>--hpx:ini</code>
-l	<code>--hpx:localities</code>
-p	<code>--hpx:app-config</code>
-q	<code>--hpx:queuing</code>
-r	<code>--hpx:run-agas-server</code>
-t	<code>--hpx:threads</code>
-v	<code>--hpx:version</code>
-w	<code>--hpx:worker</code>
-x	<code>--hpx:hpx</code>
-0	<code>--hpx:node=0</code>
-1	<code>--hpx:node=1</code>
-2	<code>--hpx:node=2</code>
-3	<code>--hpx:node=3</code>
-4	<code>--hpx:node=4</code>
-5	<code>--hpx:node=5</code>
-6	<code>--hpx:node=6</code>
-7	<code>--hpx:node=7</code>
-8	<code>--hpx:node=8</code>
-9	<code>--hpx:node=9</code>

It is possible to define your own shortcut options. In fact, all of the shortcuts listed above are pre-defined using the technique described here. Also, it is possible to redefine any of the pre-defined shortcuts to expand differently as well.

Shortcut options are obtained from the internal configuration database. They are stored as key-value properties in a special properties section named `hpx.commandline`. You can define your own shortcuts by adding the corresponding definitions to one of the `ini` configuration files as described in the section [Configuring HPX applications](#). For instance, in order to define a command line shortcut `--p` which should expand to `-hpx:print-counter`, the following configuration information needs to be added to one of the `ini` configuration files:

```
[hpx.commandline.aliases]
--pc = --hpx:print-counter
```

Note: Any arguments for shortcut options passed on the command line are retained and passed as arguments to the corresponding expanded option. For instance, given the definition above, the command line option:

```
--pc=/threads{locality#0/total}/count/cumulative
```

would be expanded to:

```
--hpx:print-counter=/threads{locality#0/total}/count/cumulative
```

Important: Any shortcut option should either start with a single '-' or with two '--' characters. Shortcuts starting with a single '-' are interpreted as short options (i.e. everything after the first character following the '-' is treated as the argument). Shortcuts starting with '--' are interpreted as long options. No other shortcut formats are supported.

Specifying options for single localities only

For runs involving more than one *locality* it is sometimes desirable to supply specific command line options to single localities only. When the *HPX* application is launched using a scheduler (like PBS, for more details see section [How to use HPX applications with PBS](#)), specifying dedicated command line options for single localities may be desirable. For this reason all of the command line options which have the general format `--hpx:<some_key>` can be used in a more general form: `--hpx:<N>:<some_key>`, where `<N>` is the number of the *locality* this command line options will be applied to, all other localities will simply ignore the option. For instance, the following PBS script passes the option `--hpx:pu-offset=4` to the *locality* '1' only.

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

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

pbsdsh -u $APP_PATH $APP_OPTIONS --hpx:1:pu-offset=4 --hpx:nodes=`cat $PBS_NODEFILE`
```

Caution: If the first application specific argument (inside `$APP_OPTIONS` is a non-option (i.e. does not start with a `-` or a `--`), then it must be placed before the option `--hpx:nodes`, which, in this case, should be the last option on the command line.

Alternatively, use the option `--hpx:endnodes` to explicitly mark the end of the list of node names:

```
pbsdsh -u $APP_PATH --hpx:1:pu-offset=4 --hpx:nodes=`cat $PBS_NODEFILE` --
↪hpx:endnodes $APP_OPTIONS
```

More details about HPX command line options

This section documents the following list of the command line options in more detail:

- *The command line option `--hpx:bind`*

The command line option `--hpx:bind`

This command line option allows one to specify the required affinity of the *HPX* worker threads to the underlying processing units. As a result the worker threads will run only on the processing units identified by the corresponding bind specification. The affinity settings are to be specified using `--hpx:bind=<BINDINGS>`, where `<BINDINGS>` have to be formatted as described below.

In addition to the syntax described below one can use `--hpx:bind=none` to disable all binding of any threads to a particular core. This is mostly supported for debugging purposes.

The specified affinities refer to specific regions within a machine hardware topology. In order to understand the hardware topology of a particular machine it may be useful to run the `lstopo` tool which is part of Portable Hardware Locality (HWLOC) to see the reported topology tree. Seeing and understanding a topology tree will definitely help in understanding the concepts that are discussed below.

Affinities can be specified using HWLOC (Portable Hardware Locality (HWLOC)) tuples. Tuples of HWLOC *objects* and associated *indexes* can be specified in the form `object:index`, `object:index-index` or `object:index, ..., index`. HWLOC objects represent types of mapped items in a topology tree. Possible

values for objects are `socket`, `numanode`, `core` and `pu` (processing unit). Indexes are non-negative integers that specify a unique physical object in a topology tree using its logical sequence number.

Chaining multiple tuples together in the more general form `object1:index1[.object2:index2[...]]` is permissible. While the first tuple's object may appear anywhere in the topology, the Nth tuple's object must have a shallower topology depth than the (N+1)th tuple's object. Put simply: as you move right in a tuple chain, objects must go deeper in the topology tree. Indexes specified in chained tuples are relative to the scope of the parent object. For example, `socket:0.core:1` refers to the second core in the first socket (all indices are zero based).

Multiple affinities can be specified using several `--hpx:bind` command line options or by appending several affinities separated by a `' '`. By default, if multiple affinities are specified, they are added.

"all" is a special affinity consisting in the entire current topology.

Note: All 'names' in an affinity specification, such as `thread`, `socket`, `numanode`, `pu` or `all` can be abbreviated. Thus the affinity specification `threads:0-3=socket:0.core:1.pu:1` is fully equivalent to its shortened form `t:0-3=s:0.c:1.p:1`.

Here is a full grammar describing the possible format of mappings:

```
mappings      ::= distribution | mapping (";" mapping) *
distribution  ::= "compact" | "scatter" | "balanced" | "numa-balanced"
mapping       ::= thread_spec "=" pu_specs
thread_spec   ::= "thread:" range_specs
pu_specs      ::= pu_spec ( "." pu_spec ) *
pu_spec       ::= type ":" range_specs | "~" pu_spec
range_specs   ::= range_spec ( "," range_spec ) *
range_spec    ::= int | int "-" int | "all"
type          ::= "socket" | "numanode" | "core" | "pu"
```

The following example assumes a system with at least 4 cores, where each core has more than 1 processing unit (hardware threads). Running `hello_world_distributed` with 4 OS-threads (on 4 processing units), where each of those threads is bound to the first processing unit of each of the cores, can be achieved by invoking:

```
hello_world_distributed -t4 --hpx:bind=thread:0-3=core:0-3.pu:0
```

Here `thread:0-3` specifies the OS threads for which to define affinity bindings, and `core:0-3.pu:0` defines that for each of the cores (`core:0-3`) only their first processing unit `pu:0` should be used.

Note: The command line option `--hpx:print-bind` can be used to print the bitmasks generated from the affinity mappings as specified with `--hpx:bind`. For instance, on a system with hyperthreading enabled (i.e. 2 processing units per core), the command line:

```
hello_world_distributed -t4 --hpx:bind=thread:0-3=core:0-3.pu:0 --hpx:print-bind
```

will cause this output to be printed:

```
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 Objects 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);
```

(continues on next page)

(continued from previous page)

```

{
    // 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
<code>hpx::future::then</code>	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 “ <code>then</code> ” 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 <code>then</code> function will help to avoid blocking waits or wasting threads on polling, greatly improving the responsiveness and scalability of an application.
un-wrap-ping con-structor for <code>hpx::future</code>	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.
<code>hpx::future::try_get</code>	There are often situations where a <code>get()</code> 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.
<code>hpx::make_ready_future</code>	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 “ <code>hpx::make_ready_future</code> ” 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 N4313¹⁴¹, *HPX* adds functions allowing to compose several futures in a more flexible way.

¹⁴¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

Table 2.14: Facilities for composing `hpx::futures`

Facility	Description	Comment
<code>hpx::when_any</code> , <code>hpx::when_any_n</code>	Asynchronously wait for at least one of multiple future or shared_future objects to finish.	N4313 ¹⁴² , ..._n versions are <i>HPX</i> only
<code>hpx::wait_any</code> , <code>hpx::wait_any_n</code>	Synchronously wait for at least one of multiple future or shared_future objects to finish.	<i>HPX</i> only
<code>hpx::when_all</code> , <code>hpx::when_all_n</code>	Asynchronously wait for all future and shared_future objects to finish.	N4313 ¹⁴³ , ..._n versions are <i>HPX</i> only
<code>hpx::wait_all</code> , <code>hpx::wait_all_n</code>	Synchronously wait for all future and shared_future objects to finish.	<i>HPX</i> only
<code>hpx::when_some</code> , <code>hpx::when_some_n</code>	Asynchronously wait for multiple future and shared_future objects to finish.	<i>HPX</i> only
<code>hpx::wait_some</code> , <code>hpx::wait_some_n</code>	Synchronously wait for multiple future and shared_future objects to finish.	<i>HPX</i> only
<code>hpx::when_each</code>	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.	<i>HPX</i> only
<code>hpx::wait_each</code> , <code>hpx::wait_each_n</code>	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.	<i>HPX</i> 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 N4409¹⁴⁴ (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411¹⁴⁵ (Task Blocks), and N4406¹⁴⁶ (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

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/2014/n4313.html>

¹⁴⁴ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf>

¹⁴⁵ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

¹⁴⁶ <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_policy` 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
<code>hpx::parallel::v1::compute_adjacent_differences</code>	Computes the differences between adjacent elements in a range.	<code><hpx/include/parallel_adjacent_find.hpp></code>	adjacent_find ¹⁴⁷
<code>hpx::parallel::v1::check_if_all_true</code>	Checks if a predicate is true for all of the elements in a range.	<code><hpx/include/parallel_all_any_none.hpp></code>	all_any_none_of ¹⁴⁸
<code>hpx::parallel::v1::check_if_any_true</code>	Checks if a predicate is true for any of the elements in a range.	<code><hpx/include/parallel_all_any_none.hpp></code>	all_any_none_of ¹⁴⁹
<code>hpx::parallel::v1::count</code>	Returns the number of elements equal to a given value.	<code><hpx/include/parallel_count.hpp></code>	count ¹⁵⁰
<code>hpx::parallel::v1::count_if</code>	Returns the number of elements satisfying a specific criteria.	<code><hpx/include/parallel_count.hpp></code>	count_if ¹⁵¹
<code>hpx::parallel::v1::determine_if_equal</code>	Determines if two sets of elements are the same.	<code><hpx/include/parallel_equal.hpp></code>	equal ¹⁵²
<code>hpx::parallel::v1::do_exclusive_scan</code>	Does an exclusive parallel scan over a range of elements.	<code><hpx/include/parallel_scan.hpp></code>	exclusive_scan ¹⁵³
<code>hpx::parallel::v1::find</code>	Finds the first element equal to a given value.	<code><hpx/include/parallel_find.hpp></code>	find ¹⁵⁴
<code>hpx::parallel::v1::find_end</code>	Finds the last sequence of elements in a certain range.	<code><hpx/include/parallel_find.hpp></code>	find_end ¹⁵⁵
<code>hpx::parallel::v1::find_first_of</code>	Searches for any one of a set of elements.	<code><hpx/include/parallel_find.hpp></code>	find_first_of ¹⁵⁶
<code>hpx::parallel::v1::find_if</code>	Finds the first element satisfying a specific criteria.	<code><hpx/include/parallel_find.hpp></code>	find ¹⁵⁷
<code>hpx::parallel::v1::find_if_not</code>	Finds the first element not satisfying a specific criteria.	<code><hpx/include/parallel_find.hpp></code>	find_if_not ¹⁵⁸
<code>hpx::parallel::v1::for_each</code>	Applies a function to a range of elements.	<code><hpx/include/parallel_for_each.hpp></code>	for_each ¹⁵⁹
<code>hpx::parallel::v1::for_each_n</code>	Applies a function to a number of elements.	<code><hpx/include/parallel_for_each.hpp></code>	for_each_n ¹⁶⁰
<code>hpx::parallel::v1::do_inclusive_scan</code>	Does an inclusive parallel scan over a range of elements.	<code><hpx/include/parallel_scan.hpp></code>	inclusive_scan ¹⁶¹
<code>hpx::parallel::v1::lexicographical_compare</code>	Checks if a range of values is lexicographically less than another range of values.	<code><hpx/include/parallel_lexicographical_compare.hpp></code>	lexicographical_compare ¹⁶²
<code>hpx::parallel::v1::mismatch</code>	Finds the first position where two ranges differ.	<code><hpx/include/parallel_mismatch.hpp></code>	mismatch ¹⁶³
<code>hpx::parallel::v1::check_if_none_true</code>	Checks if a predicate is true for none of the elements in a range.	<code><hpx/include/parallel_all_any_none.hpp></code>	all_any_none_of ¹⁶⁴
<code>hpx::parallel::v1::search</code>	Searches for a range of elements.	<code><hpx/include/parallel_search.hpp></code>	search ¹⁶⁵
<code>hpx::parallel::v1::search_n</code>	Searches for a number consecutive copies of an element in a range.	<code><hpx/include/parallel_search.hpp></code>	search_n ¹⁶⁶

¹⁴⁷ http://en.cppreference.com/w/cpp/algorithm/adjacent_find
¹⁴⁸ http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
¹⁴⁹ http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
¹⁵⁰ <http://en.cppreference.com/w/cpp/algorithm/count>
¹⁵¹ http://en.cppreference.com/w/cpp/algorithm/count_if
¹⁵² <http://en.cppreference.com/w/cpp/algorithm/equal>
¹⁵³ http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
¹⁵⁴ <http://en.cppreference.com/w/cpp/algorithm/find>
¹⁵⁵ http://en.cppreference.com/w/cpp/algorithm/find_end
¹⁵⁶ http://en.cppreference.com/w/cpp/algorithm/find_first_of
¹⁵⁷ <http://en.cppreference.com/w/cpp/algorithm/find>
¹⁵⁸ http://en.cppreference.com/w/cpp/algorithm/find_if_not
¹⁵⁹ http://en.cppreference.com/w/cpp/algorithm/for_each
¹⁶⁰ http://en.cppreference.com/w/cpp/algorithm/for_each_n
¹⁶¹ http://en.cppreference.com/w/cpp/algorithm/inclusive_scan
¹⁶² http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
¹⁶³ <http://en.cppreference.com/w/cpp/algorithm/mismatch>
¹⁶⁴ http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
¹⁶⁵ <http://en.cppreference.com/w/cpp/algorithm/search>
¹⁶⁶ 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
<code>hpx::parallel::v1::copy</code>	Copies a range of elements to a new location.	<code><hpx/include/parallel_copy.hpp></code>	exclusive_scan ¹⁶⁷
<code>hpx::parallel::v1::copy_n</code>	Copies a number of elements to a new location.	<code><hpx/include/parallel_copy.hpp></code>	copy_n ¹⁶⁸
<code>hpx::parallel::v1::copy_if</code>	Copies the elements from a range to a new location for which the given predicate is true	<code><hpx/include/parallel_copy.hpp></code>	copy ¹⁶⁹
<code>hpx::parallel::v1::move</code>	Moves a range of elements to a new location.	<code><hpx/include/parallel_fill.hpp></code>	move ¹⁷⁰
<code>hpx::parallel::v1::fill</code>	Assigns a range of elements a certain value.	<code><hpx/include/parallel_fill.hpp></code>	fill ¹⁷¹
<code>hpx::parallel::v1::fill_n</code>	Assigns a value to a number of elements.	<code><hpx/include/parallel_fill.hpp></code>	fill_n ¹⁷²
<code>hpx::parallel::v1::generate</code>	Saves the result of a function in a range.	<code><hpx/include/parallel_generate.hpp></code>	generate ¹⁷³
<code>hpx::parallel::v1::generate_n</code>	Saves the result of N applications of a function.	<code><hpx/include/parallel_generate.hpp></code>	generate_n ¹⁷⁴
<code>hpx::parallel::v1::remove</code>	Removes the elements from a range that are equal to the given value.	<code><hpx/include/parallel_remove.hpp></code>	remove ¹⁷⁵
<code>hpx::parallel::v1::remove_if</code>	Removes the elements from a range that are equal to the given predicate is false	<code><hpx/include/parallel_remove.hpp></code>	remove ¹⁷⁶
<code>hpx::parallel::v1::remove_copy</code>	Copies the elements from a range to a new location that are not equal to the given value.	<code><hpx/include/parallel_remove_copy.hpp></code>	re-move_copy ¹⁷⁷
<code>hpx::parallel::v1::remove_copy_if</code>	Copies the elements from a range to a new location for which the given predicate is false	<code><hpx/include/parallel_remove_copy.hpp></code>	re-move_copy ¹⁷⁸
<code>hpx::parallel::v1::replace</code>	Replaces all values satisfying specific criteria with another value.	<code><hpx/include/parallel_replace.hpp></code>	replace ¹⁷⁹
<code>hpx::parallel::v1::replace_if</code>	Replaces all values satisfying specific criteria with another value.	<code><hpx/include/parallel_replace.hpp></code>	replace ¹⁸⁰
<code>hpx::parallel::v1::replace_copy</code>	Copies a range, replacing elements satisfying specific criteria with another value.	<code><hpx/include/parallel_replace.hpp></code>	re-place_copy ¹⁸¹
<code>hpx::parallel::v1::replace_copy_if</code>	Copies a range, replacing elements satisfying specific criteria with another value.	<code><hpx/include/parallel_replace.hpp></code>	re-place_copy ¹⁸²
<code>hpx::parallel::v1::reverse</code>	Reverses the order elements in a range.	<code><hpx/include/parallel_reverse.hpp></code>	reverse ¹⁸³
134	Chapter 2. What's so special about HPX?		
<code>hpx::parallel::v1::reverse_copy</code>	Creates a copy of a range that is reversed.	<code><hpx/include/parallel_reverse.hpp></code>	re-verse_copy ¹⁸⁴

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

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::parallel::v1::merge</code>	Merges two sorted ranges.	<code><hpx/include/parallel_merge.hpp></code>	merge ¹⁹¹
<code>hpx::parallel::v1::inplace_merge</code>	Merges two ordered ranges in-place.	<code><hpx/include/parallel_merge.hpp></code>	inplace_merge ¹⁹²
<code>hpx::parallel::v1::includes</code>	Returns true if one set is a subset of another.	<code><hpx/include/parallel_set_operations.hpp></code>	includes ¹⁹³
<code>hpx::parallel::v1::set_difference</code>	Computes the difference between two sets.	<code><hpx/include/parallel_set_operations.hpp></code>	set_difference ¹⁹⁴
<code>hpx::parallel::v1::set_intersection</code>	Computes the intersection of two sets.	<code><hpx/include/parallel_set_operations.hpp></code>	set_intersection ¹⁹⁵
<code>hpx::parallel::v1::set_symmetric_difference</code>	Computes the symmetric difference between two sets.	<code><hpx/include/parallel_set_operations.hpp></code>	set_symmetric_difference ¹⁹⁶
<code>hpx::parallel::v1::set_union</code>	Computes the union of two sets.	<code><hpx/include/parallel_set_operations.hpp></code>	set_union ¹⁹⁷

¹⁶⁷ http://en.cppreference.com/w/cpp/algorithm/exclusive_scan

¹⁶⁸ http://en.cppreference.com/w/cpp/algorithm/copy_n

¹⁶⁹ <http://en.cppreference.com/w/cpp/algorithm/copy>

¹⁷⁰ <http://en.cppreference.com/w/cpp/algorithm/move>

¹⁷¹ <http://en.cppreference.com/w/cpp/algorithm/fill>

¹⁷² http://en.cppreference.com/w/cpp/algorithm/fill_n

¹⁷³ <http://en.cppreference.com/w/cpp/algorithm/generate>

¹⁷⁴ http://en.cppreference.com/w/cpp/algorithm/generate_n

¹⁷⁵ <http://en.cppreference.com/w/cpp/algorithm/remove>

¹⁷⁶ <http://en.cppreference.com/w/cpp/algorithm/remove>

¹⁷⁷ http://en.cppreference.com/w/cpp/algorithm/remove_copy

¹⁷⁸ http://en.cppreference.com/w/cpp/algorithm/remove_copy

¹⁷⁹ <http://en.cppreference.com/w/cpp/algorithm/replace>

¹⁸⁰ <http://en.cppreference.com/w/cpp/algorithm/replace>

¹⁸¹ http://en.cppreference.com/w/cpp/algorithm/replace_copy

¹⁸² http://en.cppreference.com/w/cpp/algorithm/replace_copy

¹⁸³ <http://en.cppreference.com/w/cpp/algorithm/reverse>

¹⁸⁴ http://en.cppreference.com/w/cpp/algorithm/reverse_copy

¹⁸⁵ <http://en.cppreference.com/w/cpp/algorithm/rotate>

¹⁸⁶ http://en.cppreference.com/w/cpp/algorithm/rotate_copy

¹⁸⁷ http://en.cppreference.com/w/cpp/algorithm/swap_ranges

¹⁸⁸ <http://en.cppreference.com/w/cpp/algorithm/transform>

¹⁸⁹ <http://en.cppreference.com/w/cpp/algorithm/unique>

¹⁹⁰ http://en.cppreference.com/w/cpp/algorithm/unique_copy

¹⁹¹ <http://en.cppreference.com/w/cpp/algorithm/merge>

¹⁹² http://en.cppreference.com/w/cpp/algorithm/inplace_merge

¹⁹³ <http://en.cppreference.com/w/cpp/algorithm/includes>

¹⁹⁴ http://en.cppreference.com/w/cpp/algorithm/set_difference

¹⁹⁵ http://en.cppreference.com/w/cpp/algorithm/set_intersection

¹⁹⁶ http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference

¹⁹⁷ 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
<code>hpx::parallel::v1::is_max_heap</code>	Returns true if the range is max heap.	<code><hpx/include/is_heap.hpp></code>	is_heap ¹⁹⁸
<code>hpx::parallel::v1::is_max_heap_until</code>	Returns the first element that breaks a max heap.	<code><hpx/include/is_heap.hpp></code>	is_heap_until ¹⁹⁹

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

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::parallel::v1::max_element</code>	Returns the largest element in a range.	<code><hpx/include/parallel_minmax.hpp></code>	max_element ²⁰⁰
<code>hpx::parallel::v1::min_element</code>	Returns the smallest element in a range.	<code><hpx/include/parallel_minmax.hpp></code>	min_element ²⁰¹
<code>hpx::parallel::v1::minmax_element</code>	Returns the smallest and the largest element in a range.	<code><hpx/include/parallel_minmax.hpp></code>	minmax_element ²⁰²

Table 2.20: Partitioning Operations (In Header: `<hpx/include/parallel_algorithm.hpp>`)

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::parallel::v1::is_partitioned</code>	Returns true if each true element for a predicate precedes the false elements in a range	<code><hpx/include/parallel_is_partitioned.hpp></code>	is_partitioned ²⁰³
<code>hpx::parallel::v1::partition</code>	Divides elements into two groups while don't preserve their relative order	<code><hpx/include/parallel_partition.hpp></code>	partition ²⁰⁴
<code>hpx::parallel::v1::partition_copy</code>	Copies a range dividing the elements into two groups	<code><hpx/include/parallel_partition.hpp></code>	partition_copy ²⁰⁵
<code>hpx::parallel::v1::stable_partition</code>	Divides elements into two groups while preserving their relative order	<code><hpx/include/parallel_partition.hpp></code>	stable_partition ²⁰⁶

¹⁹⁸ http://en.cppreference.com/w/cpp/algorithm/is_heap¹⁹⁹ http://en.cppreference.com/w/cpp/algorithm/is_heap_until²⁰⁰ http://en.cppreference.com/w/cpp/algorithm/max_element²⁰¹ http://en.cppreference.com/w/cpp/algorithm/min_element²⁰² http://en.cppreference.com/w/cpp/algorithm/minmax_element²⁰³ http://en.cppreference.com/w/cpp/algorithm/is_partitioned²⁰⁴ <http://en.cppreference.com/w/cpp/algorithm/partition>²⁰⁵ http://en.cppreference.com/w/cpp/algorithm/partition_copy²⁰⁶ 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
<code>hpx::parallel::v1::is_sorted</code>	Returns true if each element in a range is sorted	<code><hpx/include/parallel_is_sorted.hpp></code>	is_sorted ²⁰⁷
<code>hpx::parallel::v1::is_sorted_until</code>	Returns the first unsorted element	<code><hpx/include/parallel_is_sorted.hpp></code>	is_sorted_until ²⁰⁸
<code>hpx::parallel::v1::sort</code>	Sorts the elements in a range	<code><hpx/include/parallel_sort.hpp></code>	sort ²⁰⁹
<code>hpx::parallel::v1::sort_by_key</code>	Sorts one range of data using keys supplied in another range	<code><hpx/include/parallel_sort.hpp></code>	

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

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::parallel::v1::adjacent_difference</code>	Calculates the difference between each element in an input range and the preceding element.	<code><hpx/include/parallel_adjacent_difference.hpp></code>	adjacent_difference ²¹⁰
<code>hpx::parallel::v1::reduce</code>	Sums up a range of elements.	<code><hpx/include/parallel_reduce.hpp></code>	reduce ²¹¹
<code>hpx::parallel::v1::reduce_by_key</code>	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}	<code><hpx/include/parallel_reduce.hpp></code>	
<code>hpx::parallel::v1::transform_reduce</code>	Sums up a range of elements after applying a function. Also, accumulates the inner products of two input ranges.	<code><hpx/include/parallel_transform_reduce.hpp></code>	transform_reduce ²¹²
<code>hpx::parallel::v1::transform_inclusive_scan</code>	Does an inclusive parallel scan over a range of elements after applying a function.	<code><hpx/include/parallel_scan.hpp></code>	transform_inclusive_scan ²¹³
<code>hpx::parallel::v1::transform_exclusive_scan</code>	Does an exclusive parallel scan over a range of elements after applying a function.	<code><hpx/include/parallel_scan.hpp></code>	transform_exclusive_scan ²¹⁴

²⁰⁷ http://en.cppreference.com/w/cpp/algorithm/is_sorted

²⁰⁸ http://en.cppreference.com/w/cpp/algorithm/is_sorted_until

²⁰⁹ <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 en.cppreference.com
<code>hpx::parallel::v1::destroy</code>	Destroys a range of objects.	<code><hpx/include/parallel_destroy.hpp></code>	destroy ²¹⁵
<code>hpx::parallel::v1::destroy_n</code>	Destroys a range of objects.	<code><hpx/include/parallel_destroy.hpp></code>	destroy_n ²¹⁶
<code>hpx::parallel::v1::uninitialized_copy</code>	Copies a range of objects to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_copy.hpp></code>	uninitialized_copy ²¹⁷
<code>hpx::parallel::v1::uninitialized_copy_n</code>	Copies a number of objects to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_copy.hpp></code>	uninitialized_copy_n ²¹⁸
<code>hpx::parallel::v1::uninitialized_default_construct</code>	Copies a range of objects to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_default_construct.hpp></code>	uninitialized_default_construct ²¹⁹
<code>hpx::parallel::v1::uninitialized_default_construct_n</code>	Copies a number of objects to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_default_construct.hpp></code>	uninitialized_default_construct_n ²²⁰
<code>hpx::parallel::v1::uninitialized_fill</code>	Copies an object to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_fill.hpp></code>	uninitialized_fill ²²¹
<code>hpx::parallel::v1::uninitialized_fill_n</code>	Copies an object to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_fill.hpp></code>	uninitialized_fill_n ²²²
<code>hpx::parallel::v1::uninitialized_move</code>	Moves a range of objects to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_move.hpp></code>	uninitialized_move ²²³
<code>hpx::parallel::v1::uninitialized_move_n</code>	Moves a number of objects to an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_move.hpp></code>	uninitialized_move_n ²²⁴
<code>hpx::parallel::v1::uninitialized_value_construct</code>	Constructs objects in an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_value_construct.hpp></code>	uninitialized_value_construct ²²⁵
<code>hpx::parallel::v1::uninitialized_value_construct_n</code>	Constructs objects in an uninitialized area of memory.	<code><hpx/include/parallel_uninitialized_value_construct.hpp></code>	uninitialized_value_construct_n ²²⁶

²¹⁰ http://en.cppreference.com/w/cpp/algorithm/adjacent_difference²¹¹ <http://en.cppreference.com/w/cpp/algorithm/reduce>²¹² http://en.cppreference.com/w/cpp/algorithm/transform_reduce²¹³ http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan²¹⁴ 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
<code>hpx::parallel::v2::for</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/include/parallel_for_loop.hpp></code>
<code>hpx::parallel::v2::for</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/include/parallel_for_loop.hpp></code>
<code>hpx::parallel::v2::for</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/include/parallel_for_loop.hpp></code>
<code>hpx::parallel::v2::for</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/include/parallel_for_loop.hpp></code>

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

²¹⁵ <http://en.cppreference.com/w/cpp/memory/destroy>
²¹⁶ http://en.cppreference.com/w/cpp/memory/destroy_n
²¹⁷ http://en.cppreference.com/w/cpp/memory/uninitialized_copy
²¹⁸ http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n
²¹⁹ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct
²²⁰ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n
²²¹ http://en.cppreference.com/w/cpp/memory/uninitialized_fill
²²² http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n
²²³ http://en.cppreference.com/w/cpp/memory/uninitialized_move
²²⁴ http://en.cppreference.com/w/cpp/memory/uninitialized_move_n
²²⁵ http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct
²²⁶ 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 N4411²²⁷ are based on the `task_block` concept that is a part of the common subset of the [Microsoft Parallel Patterns Library \(PPL\)](https://msdn.microsoft.com/en-us/library/dd492418.aspx)²²⁸ and the [Intel Threading Building Blocks \(TBB\)](https://www.threadingbuildingblocks.org/)²²⁹ 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++](https://software.intel.com/en-us/articles/intel-cilk-plus/)²³⁰ and `async` and `finish` from [X10](https://x10-lang.org/)²³¹. 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:

²²⁷ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

²²⁸ <https://msdn.microsoft.com/en-us/library/dd492418.aspx>

²²⁹ <https://www.threadingbuildingblocks.org/>

²³⁰ <https://software.intel.com/en-us/articles/intel-cilk-plus/>

²³¹ <https://x10-lang.org/>


```

template <typename Func>
int traverse(node& n, Func && compute)
{
    int left = 0, right = 0;
    define_task_block(
        [&](task_block<>& tr) {
            if (n.left)
                tr.run([&] { left = traverse(*n.left, compute); });
            if (n.right)
                tr.run([&] { right = traverse(*n.right, compute); });
        });

    return compute(n) + left + right;
}

```

The example above demonstrates the use of two of the functions, `hpx::parallel::define_task_block` and the `hpx::parallel::task_block::run` member function of a `hpx::parallel::task_block`.

The `task_block` function delineates a region in a program code potentially containing invocations of threads spawned by the `run` member function of the `task_block` class. The `run` function spawns an *HPX* thread, a 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 N4411²³². 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.

²³² <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

An execution policy is an object that expresses the requirements on the ordering of functions invoked as a consequence of the invocation of a task block. Enabling passing an execution policy to `define_task_block` gives the user control over the amount of parallelism employed by the created `task_block`. In the following example the use of an explicit `par` execution policy makes the user's intent explicit:

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

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

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

This also causes the `hpx::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); });
            }
        });
}
```

(continues on next page)

(continued from previous page)

```

    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
```

(continues on next page)

(continued from previous page)

```
// 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.

<code>R f(p...)</code>	Synchronous Execution (returns <code>R</code>)	Asynchronous Execution (returns <code>future<R></code>)	Fire & Forget Execution (returns <code>void</code>)
Functions (direct invocation)	<code>f(p...)</code> C++	<code>async(f, p...)</code>	<code>apply(f, p...)</code>
Functions (lazy invocation)	<code>bind(f, p...)(...)</code>	<code>async(bind(f, p...), ...)</code> C++ Standard Library	<code>apply(bind(f, p...), ...)</code>
Actions (direct invocation)	<code>HPX_ACTION(f, action)</code> <code>a(id, p...)</code>	<code>HPX_ACTION(f, action)</code> <code>async(a, id, p...)</code>	<code>HPX_ACTION(f, action)</code> <code>apply(a, id, p...)</code>
Actions (lazy invocation)	<code>HPX_ACTION(f, action)</code> <code>bind(a, id, p...)</code> <code>(...)</code>	<code>HPX_ACTION(f, action)</code> <code>async(bind(a, id, p...), ...)</code>	<code>HPX_ACTION(f, action)</code> <code>apply(bind(a, id, p...), ...)</code> 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 `()()` are:

- the `id` is the global id where the final result of the asynchronous chain of operations should be sent to (in most cases this is the id of the `hpx::future` returned from the initial call to `hpx::async_continue`. Any custom continuation function should make sure this `id` is forwarded to the last operation in the chain.
- the `result` is the result value of the current operation in the asynchronous execution chain. This value needs to be forwarded to the next operation.

Note: All of those operations are implemented by the predefined continuation function object which is returned from `hpx::make_continuation`. Any (custom) function object used as a continuation should conform to the same interface.

Action error handling

Like in any other asynchronous invocation scheme it is important to be able to handle error conditions occurring while the asynchronous (and possibly remote) operation is executed. In *HPX* all error handling is based on standard C++ exception handling. Any exception thrown during the execution of an asynchronous operation will be transferred back to the original invocation *locality*, where it is rethrown during synchronization with the calling thread.

Important: Exceptions thrown during asynchronous execution can be transferred back to the invoking thread only for the synchronous and the asynchronous case with synchronization. Like with any other unhandled exception,

any exception thrown during the execution of an asynchronous action *without* synchronization will result in calling `hpx::terminate` causing the running application to exit immediately.

Note: Even if error handling internally relies on exceptions, most of the API functions exposed by *HPX* can be used without throwing an exception. Please see [Working with exceptions](#) for more information.

As an example, we will assume that the following remote function will be executed:

```
namespace app
{
    void some_function_with_error(int arg)
    {
        if (arg < 0) {
            HPX_THROW_EXCEPTION(bad_parameter, "some_function_with_error",
                               "some really bad error happened");
        }
        // do something else...
    }
}

// This will define the action type 'some_error_action' which represents
// the function 'app::some_function_with_error'.
HPX_PLAIN_ACTION(app::some_function_with_error, some_error_action);
```

The use of `HPX_THROW_EXCEPTION` to report the error encapsulates the creation of a `hpx::exception` which is initialized with the error code `hpx::bad_parameter`. Additionally it carries the passed strings, the information about the file name, line number, and call stack of the point the exception was thrown from.

We invoke this action using the synchronous syntax as described before:

```
// note: wrapped function will throw hpx::exception
some_error_action act;           // define an instance of some_error_action
try {
    act(hpx::find_here(), -3);    // exception will be rethrown from here
}
catch (hpx::exception const& e) {
    // prints: 'some really bad error happened: HPX(bad parameter)'
    cout << e.what();
}
```

If this action is invoked asynchronously with synchronization, the exception is propagated to the waiting thread as well and is re-thrown from the future's function `get()`:

```
// note: wrapped function will throw hpx::exception
some_error_action act;           // define an instance of some_error_action
hpx::future<void> f = hpx::async(act, hpx::find_here(), -3);
try {
    f.get();                      // exception will be rethrown from here
}
catch (hpx::exception const& e) {
    // prints: 'some really bad error happened: HPX(bad parameter)'
    cout << e.what();
}
```

For more information about error handling please refer to the section [Working with exceptions](#). There we also explain how to handle error conditions without having to rely on exception.

Writing components

A component in *HPX* is a C++ class which can be created remotely and for which its member functions can be invoked remotely as well. The following sections highlight how components can be defined, created, and used.

Defining components

In order for a C++ class type to be managed remotely in *HPX*, the type must be derived from the `hpx::components::component_base` template type. We call such C++ class types ‘components’.

Note that the component type itself is passed as a template argument to the base class:

```
// header file some_component.hpp

#include <hpx/include/components.hpp>

namespace app
{
    // Define a new component type 'some_component'
    struct some_component
    : hpx::components::component_base<some_component>
    {
        // This member function is has to be invoked remotely
        int some_member_function(std::string const& s)
        {
            return boost::lexical_cast<int>(s);
        }

        // This will define the action type 'some_member_action' which
        // represents the member function 'some_member_function' of the
        // object type 'some_component'.
        HPX_DEFINE_COMPONENT_ACTION(some_component, some_member_function, some_member_
↪action);
    };
}

// This will generate the necessary boiler-plate code for the action allowing
// it to be invoked remotely. This declaration macro has to be placed in the
// header file defining the component itself.
//
// Note: The second argument to the macro below has to be systemwide-unique
//       C++ identifiers
//
HPX_REGISTER_ACTION_DECLARATION(app::some_component::some_member_action, some_
↪component_some_action);
```

There is more boiler plate code which has to be placed into a source file in order for the component to be usable. Every component type is required to have macros placed into its source file, one for each component type and one macro for each of the actions defined by the component type.

For instance:

```
// source file some_component.cpp

#include "some_component.hpp"

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

(continues on next page)

(continued from previous page)

```
// remote creation of 'app::some_component' instances with 'hpx::new_<>()'
//
using some_component = app::some_component;
using some_component_type = hpx::components::component<some_component>;

// Please note that the second argument to this macro must be a
// (system-wide) unique C++-style identifier (without any namespaces)
//
HPX_REGISTER_COMPONENT(some_component_type, some_component);

// The parameters for this macro have to be the same as used in the corresponding
// HPX_REGISTER_ACTION_DECLARATION() macro invocation in the corresponding
// header file.
//
// Please note that the second argument to this macro must be a
// (system-wide) unique C++-style identifier (without any namespaces)
//
HPX_REGISTER_ACTION(app::some_component::some_member_action, some_component_some_
↪action);
```

Defining client side representation classes

Often it is very convenient to define a separate type for a component which can be used on the client side (from where the component is instantiated and used). This step might seem as unnecessary duplicating code, however it significantly increases the type safety of the code.

A possible implementation of such a client side representation for the component described in the previous section could look like:

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

namespace app
{
    // Define a client side representation type for the component type
    // 'some_component' defined in the previous section.
    //
    struct some_component_client
    : hpx::components::client_base<some_component_client, some_component>
    {
        using base_type = hpx::components::client_base<
            some_component_client, some_component>;

        some_component_client(hpx::future<hpx::id_type> && id)
            : base_type(std::move(id))
        {}

        hpx::future<int> some_member_function(std::string const& s)
        {
            some_component::some_member_action act;
            return hpx::async(act, get_id(), s);
        }
    };
}
```

A client side object stores the global id of the component instance it represents. This global id is accessible by calling the function `client_base<>::get_id()`. The special constructor which is provided in the example allows to

create this client side object directly using the API function `hpx::new_`.

Creating component instances

Instances of defined component types can be created in two different ways. If the component to create has a defined client side representation type, then this can be used, otherwise use the server type.

The following examples assume that `some_component_type` is the type of the server side implementation of the component to create. All additional arguments (see , ... notation below) are passed through to the corresponding constructor calls of those objects:

```
// create one instance on the given locality
hpx::id_type here = hpx::find_here();
hpx::future<hpx::id_type> f =
    hpx::new_<some_component_type>(here, ...);

// create one instance using the given distribution
// policy (here: hpx::colocating_distribution_policy)
hpx::id_type here = hpx::find_here();
hpx::future<hpx::id_type> f =
    hpx::new_<some_component_type>(hpx::colocated(here), ...);

// create multiple instances on the given locality
hpx::id_type here = find_here();
hpx::future<std::vector<hpx::id_type>> f =
    hpx::new_<some_component_type[]>(here, num, ...);

// create multiple instances using the given distribution
// policy (here: hpx::binpacking_distribution_policy)
hpx::future<std::vector<hpx::id_type>> f = hpx::new_<some_component_type[]>(
    hpx::binpacking(hpx::find_all_localities()), num, ...);
```

The examples below demonstrate the use of the same API functions for creating client side representation objects (instead of just plain ids). These examples assume that `client_type` is the type of the client side representation of the component type to create. As above, all additional arguments (see , ... notation below) are passed through to the corresponding constructor calls of the server side implementation objects corresponding to the `client_type`:

```
// create one instance on the given locality
hpx::id_type here = hpx::find_here();
client_type c = hpx::new_<client_type>(here, ...);

// create one instance using the given distribution
// policy (here: hpx::colocating_distribution_policy)
hpx::id_type here = hpx::find_here();
client_type c = hpx::new_<client_type>(hpx::colocated(here), ...);

// create multiple instances on the given locality
hpx::id_type here = hpx::find_here();
hpx::future<std::vector<client_type>> f =
    hpx::new_<client_type[]>(here, num, ...);

// create multiple instances using the given distribution
// policy (here: hpx::binpacking_distribution_policy)
hpx::future<std::vector<client_type>> f = hpx::new_<client_type[]>(
    hpx::binpacking(hpx::find_all_localities()), num, ...);
```

Using component instances

Segmented containers

In parallel programming, there is now a plethora of solutions aimed at implementing “partially contiguous” or segmented data structures, whether on shared memory systems or distributed memory systems. *HPX* implements such structures by drawing inspiration from Standard C++ containers.

Using segmented containers

A segmented container is a template class that is described in the namespace `hpx`. All segmented containers are very similar semantically to their sequential counterpart (defined in namespace `std` but with an additional template parameter named `DistPolicy`). The distribution policy is an optional parameter that is passed last to the segmented container constructor (after the container size when no default value is given, after the default value if not). The distribution policy describes the manner in which a container is segmented and the placement of each segment among the available runtime localities.

However, only a part of the `std` container member functions were reimplemented:

- (constructor), (destructor), operator=
- operator[]
- begin, cbegin, end, cend
- size

An example of how to use the `partitioned_vector` container would be:

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

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

// By default, the number of segments is equal to the current number of
// localities
//
hpx::partitioned_vector<double> va(50);
hpx::partitioned_vector<double> vb(50, 0.0);
```

An example of how to use the `partitioned_vector` container with distribution policies would be:

```
#include <hpx/include/partitioned_vector.hpp>
#include <hpx/runtime/find_localities.hpp>

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

std::size_t num_segments = 10;
std::vector<hpx::id_type> locs = hpx::find_all_localities()

auto layout =
    hpx::container_layout( num_segments, locs );
```

(continues on next page)

(continued from previous page)

```
// The number of segments is 10 and those segments are spread across the
// localities collected in the variable locs in a Round-Robin manner
//
hpx::partitioned_vector<double> va(50, layout);
hpx::partitioned_vector<double> vb(50, 0.0, layout);
```

By definition, a segmented container must be accessible from any thread although its construction is synchronous only for the thread who has called its constructor. To overcome this problem, it is possible to assign a symbolic name to the segmented container:

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

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

hpx::future<void> fserver = hpx::async(
    [] () {
        hpx::partitioned_vector<double> v(50);

        // Register the 'partitioned_vector' with the name "some_name"
        //
        v.register_as("some_name");

        /* Do some code */
    });

hpx::future<void> fclient =
    hpx::async(
        [] () {
            // Naked 'partitioned_vector'
            //
            hpx::partitioned_vector<double> v;

            // Now the variable v points to the same 'partitioned_vector' that has
            // been registered with the name "some_name"
            //
            v.connect_to("some_name");

            /* Do some code */
        });
```

Segmented containers

HPX provides the following segmented containers:

Table 2.25: Sequence containers

Name	Description	In header	Class page at cppreference.com
<code>hpx::partitioned_vector</code>	Dynamic segmented contiguous array.	<code><hpx/include/partitioned_vector.hpp></code>	vector ²³³

Table 2.26: Unordered associative containers

Name	Description	In header	Class page at cp-preference.com
<code>hpx::unordered_map</code>	Segmented collection of key-value pairs, hashed by keys, keys are unique.	<code><hpx/include/unordered_map.hpp></code>	unordered_map ²³⁴

Segmented iterators and segmented iterator traits

The basic iterator used in the STL library is only suitable for one-dimensional structures. The iterators we use in *HPX* must adapt to the segmented format of our containers. Our iterators are then able to know when incrementing themselves if the next element of type `T` is in the same data segment or in another segment. In this second case, the iterator will automatically point to the beginning of the next segment.

Note: Note that the dereference operation `operator *` does not directly return a reference of type `T&` but an intermediate object wrapping this reference. When this object is used as an l-value, a remote write operation is performed; When this object is used as an r-value, implicit conversion to `T` type will take care of performing remote read operation.

It is sometimes useful not only to iterate element by element, but also segment by segment, or simply get a local iterator in order to avoid additional construction costs at each dereferencing operations. To mitigate this need, the `hpx::traits::segmented_iterator_traits` are used.

With `segmented_iterator_traits` users can uniformly get the iterators which specifically iterates over segments (by providing a segmented iterator as a parameter), or get the local begin/end iterators of the nearest local segment (by providing a per-segment iterator as a parameter):

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

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

using iterator = hpx::partitioned_vector<T>::iterator;
using traits   = hpx::traits::segmented_iterator_traits<iterator>;

hpx::partitioned_vector<T> v;
std::size_t count = 0;

auto seg_begin = traits::segment(v.begin());
auto seg_end   = traits::segment(v.end());

// Iterate over segments
for (auto seg_it = seg_begin; seg_it != seg_end; ++seg_it)
{
    auto loc_begin = traits::begin(seg_it)
    auto loc_end   = traits::end(seg_it);

    // Iterate over elements inside segments
    for (auto lit = loc_begin; lit != loc_end; ++lit, ++count)
```

(continues on next page)

²³³ <http://en.cppreference.com/w/cpp/container/vector>

²³⁴ http://en.cppreference.com/w/cpp/container/unordered_map

(continued from previous page)

```
{
    *lit = count;
}
}
```

Which is equivalent to:

```
hpx::partitioned_vector<T> v;
std::size_t count = 0;

auto begin = v.begin();
auto end   = v.end();

for (auto it = begin; it != end; ++it, ++count)
{
    *it = count;
}
```

Using views

The use of multidimensional arrays is quite common in the numerical field whether to perform dense matrix operations or to process images. It exist many libraries which implement such object classes overloading their basic operators (e.g. “+”, “-”, “*”, “()”, etc.). However, such operation becomes more delicate when the underlying data layout is segmented or when it is mandatory to use optimized linear algebra subroutines (i.e. BLAS subroutines).

Our solution is thus to relax the level of abstraction by allowing the user to work not directly on n-dimensionnal data, but on “n-dimensionnal collections of 1-D arrays”. The use of well-accepted techniques on contiguous data is thus preserved at the segment level, and the composability of the segments is made possible thanks to multidimensional array-inspired access mode.

Preface: Why SPMD?

Although *HPX* refutes by design this programming model, the *locality* plays a dominant role when it comes to implement vectorized code. To maximize local computations and avoid unneeded data transfers, a parallel section (or Single Programming Multiple Data section) is required. Because the use of global variables is prohibited, this parallel section is created via the RAII idiom.

To define a parallel section, simply write an action taking a `spmd_block` variable as a first parameter:

```
#include <hpx/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;

    // Instantiate the parallel section
    hpx::lcos::define_spmd_block(
        "some_name", images_per_locality, std::move(act) /*, arg0, arg1, ... */);

    return 0;
}
```

Note: In principle, the user should never call the `spmd_block` constructor. The `define_spmd_block` function is responsible of instantiating `spmd_block` objects and broadcasting them to each created image.

SPMD multidimensional views

Some classes are defined as “container views” when the purpose is to observe and/or modify the values of a container using another perspective than the one that characterizes the container. For example, the values of an `std::vector` object can be accessed via the expression `[i]`. Container views can be used, for example, when it is desired for those values to be “viewed” as a 2D matrix that would have been flattened in a `std::vector`. The values would be possibly accessible via the expression `vv(i, j)` which would call internally the expression `v[k]`.

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

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

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

using iterator = hpx::partitioned_vector<double>::iterator;
using traits = hpx::traits::segmented_iterator_traits<iterator>;

hpx::partitioned_vector<double> v;

// Create a 1-D view of the vector of segments
auto vv = traits::segment(v.begin());

// Access segment i
std::vector<double> v = vv[i];
```

Our views are called “multidimensional” in the sense that they generalize to N dimensions the purpose of `segmented_iterator_traits::segment()` in the 1-D case. Note that in a parallel section, the 2-D expression `a(i, j) = b(i, j)` is quite confusing because without convention, each of the images invoked will race

to execute the statement. For this reason, our views are not only multidimensional but also “spmd-aware”.

Note: SPMD-awareness: The convention is simple. If an assignment statement contains a view subscript as an l-value, it is only and only the image holding the r-value who is evaluating the statement. (In MPI sense, it is called a Put operation).

Subscript-based operations

Here are some examples of using subscripts in the 2-D view case:

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

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

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

/* Do some code */

Vec v;

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

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

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

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

Iterator-based operations

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

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

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

(continues on next page)

(continued from previous page)

```

using Vec = hpx::partitioned_vector<int>;
using View_3D = hpx::partitioned_vector_view<int, 3>;

/* Do some code */

Vec v1, v2;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t size_x, size_y, size_z;

    // Instanciate the views
    View_3D vv1(block, v1.begin(), v1.end(), {size_x, size_y, size_z});
    View_3D vv2(block, v2.begin(), v2.end(), {size_x, size_y, size_z});

    // Save previous segments covered by vv1 into segments covered by vv2
    auto vv2_it = vv2.begin();
    auto vv1_it = vv1.cbegin();

    for(; vv2_it != vv2.end(); vv2_it++, vv1_it++)
    {
        // It's a Put operation
        *vv2_it = *vv1_it;
    }

    // Ensure that all images have performed their Put operations
    block.sync_all();

    // Ensure that only one image is putting updated data into the different
    // segments covered by vv1
    if(block.this_image() == 0)
    {
        int idx = 0;

        // Update all the segments covered by vv1
        for(auto i = vv1.begin(); i != vv1.end(); i++)
        {
            // It's a Put operation
            *i = std::vector<float>(elt_size, idx++);
        }
    }
}

```

Here is an example that shows how to iterate only over segments owned by the current image:

```

#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/components/containers/partitioned_vector/partitioned_vector_local_view.
↪hpp>
#include <hpx/include/partitioned_vector.hpp>

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

using Vec = hpx::partitioned_vector<float>;

```

(continues on next page)

(continued from previous page)

```

using View_1D = hpx::partitioned_vector_view<float,1>;

/* Do some code */

Vec v;

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

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

    // Instantiate the local view from the view
    auto local_vv = hpx::local_view(vv);

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

        /* Do some code */
    }
}

```

Instantiating sub-views

It is possible to construct views from other views: we call it sub-views. The constraint nevertheless for the subviews is to retain the dimension and the value type of the input view. Here is an example showing how to create a sub-view:

```

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

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

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

/* Do some code */

Vec v;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t N = 20;
    std::size_t tileSize = 5;

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

    // Instantiate the subview
    View_2D svv(

```

(continues on next page)

(continued from previous page)

```

        block, &vv(tilesz, 0), &vv(2*tilesz-1, tilesz-1), {tilesz, tilesz}, {N, N});

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

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

```

Note: The last parameter of the subview constructor is the size of the original view. If one would like to create a subview of the subview and so on, this parameter should stay unchanged. {N, N} for the above example).

C++ co-arrays

Fortran has extended its scalar element indexing approach to reference each segment of a distributed array. In this extension, a segment is attributed a ?co-index? and lives in a specific *locality*. A co-index provides the application with enough information to retrieve the corresponding data reference. In C++, containers present themselves as a ?smarter? alternative of Fortran arrays but there are still no corresponding standardized features similar to the Fortran co-indexing approach. We present here an implementation of such features in *HPX*.

Preface: co-array, a segmented container tied to a SPMD multidimensional views

As mentioned before, a co-array is a distributed array whose segments are accessible through an array-inspired access mode. We have previously seen that it is possible to reproduce such access mode using the concept of views. Nevertheless, the user must pre-create a segmented container to instantiate this view. We illustrate below how a single constructor call can perform those two operations:

```

#include <hpx/components/containers/coarray/coarray.hpp>
#include <hpx/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
<code>block</code>	Reference to a <code>spmd_block</code> object
<code>"a"</code>	Symbolic name of type <code>std::string</code>
<code>{height,width, _}</code>	Dimensions of the coarray object
<code>segment_size</code>	Size of a co-indexed element (i.e. size of the object referenced by the expression <code>a(i, j, k)</code>)

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++)
```

(continues on next page)

(continued from previous page)

```

{
    // Local write operation performed via the use of local subscript
    a(i,j,_) = std::vector<double>(elt_size,idx);
    idx++;
}

block.sync_all();
}

```

Note: When the “last dimension index” of a subscript is equal to `hpx::container::placeholders::_`, local subscript (and not global subscript) is used. It is equivalent to a global subscript used with a “last dimension index” equal to the value returned by `block.this_image()`.

2.5.8 Running on batch systems

This section walks you through launching *HPX* applications on various batch systems.

How to use *HPX* applications with PBS

Most *HPX* applications are executed on parallel computers. These platforms typically provide integrated job management services that facilitate the allocation of computing resources for each parallel program. *HPX* includes 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](#)²³⁵ for more information), below is a minimal example to illustrate the approach. As a test application we will use the multithreaded `hello_world_distributed` program, explained in the section *Remote execution with actions: Hello world*.

```

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

APP_PATH=~/.packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

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

```

Caution: If the first application specific argument (inside `$APP_OPTIONS`) is a non-option (i.e. does not start with a `-` or a `--`), then 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
```

²³⁵ <http://www.clusterresources.com/torquedocs21/>

The `#PBS -l nodes=2:ppn=4` directive will cause two compute nodes to be allocated for the application, as specified in the option `nodes`. Each of the nodes will dedicate four cores to the program, as per the option `ppn`, short for “processors per node” (PBS does not distinguish between processors and cores). Note that requesting more cores per node than physically available is pointless and may prevent PBS from accepting the script.

On newer PBS versions the PBS command syntax might be different. For instance, the PBS script above would look like:

```
#!/bin/bash
#
#PBS -l select=2:ncpus=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

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

`APP_PATH` and `APP_OPTIONS` are shell variables that respectively specify the correct path to the executable (`hello_world_distributed` in this case) and the command line options. Since the `hello_world_distributed` application doesn’t need any command line options, `APP_OPTIONS` has been left empty. Unlike in other execution environments, there is no need to use the `--hpx:threads` option to indicate the required number of OS threads per node; the *HPX* library will derive this parameter automatically from PBS.

Finally, `pbsdsh` is a PBS command that starts tasks to the resources allocated to the current job. It is recommended to leave this line as shown and modify only the PBS options and shell variables as needed for a specific application.

Important: A script invoked by `pbsdsh` starts in a very basic environment: the user’s `$HOME` directory is defined and is the current directory, the `LANG` variable is set to `C` and the `PATH` is set to the basic `/usr/local/bin:/usr/bin:/bin` as defined in a system-wide file `pbs_environment`. Nothing that would normally be set up by a system shell profile or user shell profile is defined, unlike the environment for the main job script.

Another choice is for the `pbsdsh` command in your main job script to invoke your program via a shell, like `sh` or `bash` so that it gives an initialized environment for each instance. 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_distributed
APP_OPTIONS=

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

All that remains now is submitting the job to the queuing system. Assuming that the contents of the PBS script were saved in 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 *sbatch* directly. The parameters are the same as with *run*. The commands you need to execute are the same you would need to start your application as if you were in an interactive shell.

2.5.9 Debugging *HPX* applications

Using a debugger with *HPX* applications

Using a debugger such as *gdb* with *HPX* applications is no problem. However, there are some things to keep in mind to make the experience somewhat more productive.

Call stacks in *HPX* can often be quite unwieldy as the library is heavily templated and the call stacks can be very deep. For this reason it is sometimes a good idea compile *HPX* in `RelWithDebInfo` mode which applies some optimizations but keeps debugging symbols. This can often compress call stacks significantly. On the other hand, stepping through the code can also be more difficult because of statements being reordered and variables being optimized away. Also note that because *HPX* implements user-space threads and context switching, call stacks may not always be complete in a debugger.

HPX launches not only worker threads but also a few helper threads. The first thread is the main thread which typically does no work in an *HPX* application, except at startup and shutdown. If using the default settings, *HPX* will spawn six

additional threads (used for service thread pools). The first worker thread is usually the eighth thread, and most user 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

Wild-card	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

Wild-card	Description
*	This wildcard character matches any <i>locality</i> or any thread, depending on whether it is used for <code>locality#*</code> or <code>worker-thread#*</code> . 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
<code>--hpx:print-counter</code>	print the specified performance counter either repeatedly and/or at the times specified by <code>--hpx:print-counter-at</code> (see also option <code>--hpx:print-counter-interval</code>).
<code>--hpx:print-counter-reset</code>	print the specified performance counter either repeatedly and/or at the times specified by <code>--hpx:print-counter-at</code> reset the counter after the value is queried. (see also option <code>--hpx:print-counter-interval</code>).
<code>--hpx:print-counter-interval</code>	print the performance counter(s) specified with <code>--hpx:print-counter</code> repeatedly after the time interval (specified in milliseconds) (default:0 which means print once at shutdown).
<code>--hpx:print-counter-file</code>	print the performance counter(s) specified with <code>--hpx:print-counter</code> to the given file (default: console).
<code>--hpx:list-counter-names</code>	list the names of all registered performance counters.
<code>--hpx:list-counter-descriptions</code>	list the description of all registered performance counters.
<code>--hpx:print-counter-format</code>	print the performance counter(s) specified with <code>--hpx:print-counter</code> possible formats in csv format with header or without any header (see option <code>--hpx:no-csv-header</code>), possible values: <code>csv</code> (prints counter values in CSV format with full names as header) <code>csv-short</code> (prints counter values in CSV format with shortnames provided with <code>--hpx:print-counter</code> as <code>--hpx:print-counter shortname,full-countername</code>)
<code>--hpx:no-csv-header</code>	print the performance counter(s) specified with <code>--hpx:print-counter</code> and <code>csv</code> or <code>csv-short</code> format specified with <code>--hpx:print-counter-format</code> without header.
<code>--hpx:print-counter-when</code> arg	print the performance counter(s) specified with <code>--hpx:print-counter</code> (or <code>--hpx:print-counter-reset</code>) at the given point in time, possible argument values: <code>startup</code> , <code>shutdown</code> (default), <code>noshutdown</code> .
<code>--hpx:reset-counter</code>	reset all performance counter(s) specified with <code>--hpx:print-counter</code> 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_distributed` example.

Listing all available counters `hello_world_distributed --hpx:list-counters` yields:

```
List of available counter instances (replace * below with the appropriate
sequence number)
-----
/agas/count/allocate /agas/count/bind /agas/count/bind_gid
/agas/count/bind_name ... /threads{locality#*/allocator#*/count/objects
/threads{locality#*/total}/count/stack-recycles
/threads{locality#*/total}/idle-rate
/threads{locality#*/worker-thread#*/idle-rate
```

Providing more information about all available counters `hello_world_distributed --hpx:list-counter-infos` yields:

```
Information about available counter instances (replace * below with the
appropriate sequence number)
-----
fullname: /agas/count/allocate helptext: returns the number of invocations of
the AGAS service 'allocate' type: counter_raw version: 1.0.0
-----
-----
fullname: /agas/count/bind helptext: returns the number of invocations of the
AGAS service 'bind' type: counter_raw version: 1.0.0
-----
-----
fullname: /agas/count/bind_gid helptext: returns the number of invocations of
the AGAS service 'bind_gid' type: counter_raw version: 1.0.0
-----
...

```

This command will not only list the counter names but also a short description of the data exposed by this counter.

Note: The list of available counters may differ depending on the concrete execution environment (hardware or software) of your application.

Requesting the counter data for one or more performance counters can be achieved by invoking `hello_world_distributed` with a list of counter names:

```
hello_world_distributed \
--hpx:print-counter=/threads{locality#0/total}/count/cumulative \
--hpx:print-counter=/agas{locality#0/total}/count/bind
```

which yields for instance:

```
hello world from OS-thread 0 on locality 0
/threads{locality#0/total}/count/cumulative,1,0.212527,[s],33
/agas{locality#0/total}/count/bind,1,0.212790,[s],11
```

The first line is the normal output generated by `hello_world_distributed` and has no relation to the counter data listed. The last two lines contain the counter data as gathered at application shutdown. These lines have 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_distributed \
  --hpx:print-counter=/threads{locality#0/total}/count/cumulative \
  --hpx:print-counter=/agas{locality#0/total}/count/bind \
  --hpx:print-counter-interval=20
```

yields for instance (leaving off the actual console output of the `hello_world_distributed` example for brevity):

```
threads{locality#0/total}/count/cumulative,1,0.002409,[s],22
agas{locality#0/total}/count/bind,1,0.002542,[s],9
threads{locality#0/total}/count/cumulative,2,0.023002,[s],41
agas{locality#0/total}/count/bind,2,0.023557,[s],10
threads{locality#0/total}/count/cumulative,3,0.037514,[s],46
agas{locality#0/total}/count/bind,3,0.038679,[s],10
```

The command `--hpx:print-counter-destination=<file>` will redirect all counter data gathered to the specified file name, which avoids cluttering the console output of your application.

The command line option `--hpx:print-counter` supports using a limited set of wildcards for a (very limited) set of use cases. In particular, all occurrences of `#*` as in `locality#*` and in `worker-thread#*` will be automatically expanded to the proper set of performance counter names representing the actual environment for the executed program. For instance, if your program is utilizing 4 worker threads for the execution of HPX threads (see command line option `--hpx:threads`) the following command line

```
hello_world_distributed \
  --hpx:threads=4 \
  --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative
```

will print the value of the performance counters monitoring each of the worker threads:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 0
hello world from OS-thread 2 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.0025214,[s],27
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.0025453,[s],33
/threads{locality#0/worker-thread#2}/count/cumulative,1,0.0025683,[s],29
/threads{locality#0/worker-thread#3}/count/cumulative,1,0.0025904,[s],33
```

The command `--hpx:print-counter-format` takes values `csv` and `csv-short` to generate CSV formatted counter values with header.

With format as `csv`:

```
hello_world_distributed \  
  --hpx:threads=2 \  
  --hpx:print-counter-format csv \  
  --hpx:print-counter /threads{locality#*/total}/count/cumulative \  
  --hpx:print-counter /threads{locality#*/total}/count/cumulative-phases
```

will print the values of performance counters in CSV format with 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_distributed \  
  --hpx:threads 2 \  
  --hpx:print-counter-format csv-short \  
  --hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \  
  --hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases
```

will print the values of performance counters in CSV format with 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_distributed \  
  --hpx:threads 2 \  
  --hpx:print-counter-format csv-short \  
  --hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \  
  --hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases \  
  --hpx:print-counter-interval 5
```

will print the header only once repeating the performance counter value(s) repeatedly:

```
cum,phases  
25,42  
hello world from OS-thread 1 on locality 0  
hello world from OS-thread 0 on locality 0  
44,95
```

The command `--hpx:no-csv-header` to be used with `--hpx:print-counter-format` to print performance counter values in CSV format without any header:

```
hello_world_distributed \  
--hpx:threads 2 \  
--hpx:print-counter-format csv-short \  
--hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \  
--hpx:print-counter-interval 5
```

(continues on next page)

(continued from previous page)

```
--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 are two ways of exposing your own Performance Counter data: a simple, function based way and a more complex, but more powerful way of implementing a full Performance Counter. Both alternatives are described in the following sections.

Exposing performance counter data using a simple function

The simplest way to expose arbitrary numeric data is to write a function which will then be called whenever a consumer queries this counter. Currently, this type of Performance Counter can only be used to expose integer values. The expected signature of this function is:

```
std::int64_t some_performance_data(bool reset);
```

The argument `bool reset` (which is supplied by the runtime system when the function is invoked) specifies whether the counter value should be reset after evaluating the current value (if applicable).

For instance, here is such a function returning how often it was invoked:

```
// The atomic variable 'counter' ensures the thread safety of the counter.
boost::atomic<std::int64_t> counter(0);

std::int64_t some_performance_data(bool reset)
{
    std::int64_t result = ++counter;
    if (reset)
        counter = 0;
    return result;
}
```

This example function exposes a linearly increasing value as our performance data. The value is incremented on each invocation, 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
```

(continues on next page)

(continued from previous page)

```

        "returns a linearly increasing counter value"    // description text (optional)
        ""                                              // unit of measure (optional)
    );
}

```

Now it is possible to instantiate a new counter instance based on the naming scheme `"/test{locality#*/total}/data"` where `*` is a zero based integer index identifying the *locality* for which the counter instance should be accessed. The function `hpx::performance_counters::install_counter_type` enables 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)

```

(continues on next page)

(continued from previous page)

```

#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);
    }
}
}

```

(continues on next page)

(continued from previous page)

```

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
    {

```

(continues on next page)

(continued from previous page)

```

        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_counters/base_performance_counter.hpp]` 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

```

(continues on next page)

(continued from previous page)

```

        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.

```

(continues on next page)

(continued from previous page)

```

    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](http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern)²³⁶. This base class itself is derived from the `performance_counter` interface described above.

Additionally, a full Performance Counter implementation not only exposes the actual value but also provides information about

- The point in time a particular value was retrieved
- A (sequential) invocation count
- The actual counter value
- An optional scaling coefficient
- Information about the counter status

Existing HPX performance counters

The HPX runtime system exposes a wide variety of predefined Performance Counters. These counters expose critical information about different modules of the runtime system. They can help determine system bottlenecks and fine-tune system and application performance.

²³⁶ http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Table 2.31: AGAS performance counters

Counter type	Counter instance formatting	Description	Parameters
<code>/agas/count/ <agas_service></code> where: <code><agas_service></code> is one of the following: <i>primary</i> <i>namespace</i> <i>pace</i> <i>services:</i> route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate, begin_migration, end_migration <i>component</i> <i>namespace</i> <i>pace</i> <i>services:</i> bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_type, num_localities_type <i>locality</i> <i>namespace</i> <i>services:</i> free, localities, num_localities, num_threads, resolve_locality, resolved_localities <i>symbol</i> <i>namespace</i> <i>services:</i> bind, resolve, unbind, iterate_names, on_symbol_namespace_event	<code><agas_instance>/total</code> where: <code><agas_instance></code> is the name of the <i>AGAS</i> service to query. Currently, this value will be <i>locality#0</i> where 0 is the root <i>locality</i> (the id of the <i>locality</i> hosting the <i>AGAS</i> service). The value for * can be any <i>locality</i> id for the following <code><agas_service></code> : route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, bin, resolve, unbind, and iterate_names only the primary and symbol <i>AGAS</i> service components live on all localities, whereas all other <i>AGAS</i> services are available on <i>locality#0</i> only).	None	Returns the total number of invocations of the specified <i>AGAS</i> service since its creation.
<code>/agas/ <agas_service_category>/total count</code> where: <code><agas_service_category></code> is one of the following: primary , locality , component or symbol	<code><agas_instance>/total</code> where: <code><agas_instance></code> is the name of the <i>AGAS</i> service to query. Currently, this value will be <i>locality#0</i> where 0 is the root <i>locality</i> (the id of the <i>locality</i> hosting the <i>AGAS</i> service). Except for <code><agas_service_category></code> , <i>primary</i> or <i>symbol</i> for which the value for * can be any <i>locality</i> id (only the primary and symbol <i>AGAS</i> service components live on all localities, whereas all other <i>AGAS</i> services are available on	None	Returns the overall total number of invocations of all <i>AGAS</i> services provided by the given <i>AGAS</i> service category since its creation.
2.5. Manual			185

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 <i>locality</i> id of the <i>locality</i> the overall number of transmitted bytes should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall number of raw (uncompressed) bytes sent or received (see <operation>, e.g. sent or received) for the specified <connection_type>. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPOR_T_MPI was defined while compiling the <i>HPX</i> core library (which is not defined by default, the corresponding cmake configuration constant is HPX_WITH_PARCELPOR_T_MPI. Please see <i>CMake variables used to configure HPX</i> 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 <i>locality</i> id of the <i>locality</i> the total transmission time should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	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 <i>locality</i> (see <operation>, e.g. sent or received). The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPOR_T_MPI was defined while compiling the <i>HPX</i> core library (which is not defined by default, the corresponding cmake configuration constant is HPX_WITH_PARCELPOR_T_MPI. Please see <i>CMake variables used to configure HPX</i> 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 <i>locality</i> id of the <i>locality</i> the overall number of transmitted bytes should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall number of bytes transferred (see <operation>, e.g. sent or received possibly compressed) for the specified <connection_type> by the given <i>locality</i> . The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPOR_T_MPI was defined while compiling the <i>HPX</i> core library (which is not defined by default, the corresponding cmake configuration constant is HPX_WITH_PARCELPOR_T_MPI. Please see <i>CMake variables used to configure HPX</i> for more details.	If the configure-time option -DHPX_WITH_PARCELPOR_T_ACTION 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 <i>locality</i> id of the	Returns the overall time spent performing outgoing data serialization for the specified <connection_type> on the given <i>locality</i> (see <operation>, e.g. sent or received). The performance counters for the connection	If the configure-time option -DHPX_WITH_PARCELPOR_T_ACTION was specified, this counter allows to

²³⁷ 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	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where: locality#* is defining the <i>locality</i> for which the overall number of retired <i>HPX</i>-threads should be queried for. The <i>locality</i> id (given by * is a (zero based) number identifying the <i>locality</i>.</p> <p>pool#* is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p>worker-thread#* is defining the worker thread for which the overall number of retired <i>HPX</i>-threads should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the 'default' pool.</p>	<p>Returns the overall number of executed (retired) <i>HPX</i>-threads on the given <i>locality</i> since application start. If the instance name is <code>total</code> the counter returns the accumulated number of retired <i>HPX</i>-threads for all worker threads (cores) on that <i>locality</i>. If the instance name is <code>worker-thread#*</code> the counter will return the overall number of retired <i>HPX</i>-threads for all worker threads separately.</p> <p>The current value of the idle-loop counter should be queried for.</p> <p>The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the 'default' pool.</p>	<p>None</p> <p>HPX_WITH_THREAD_CUMULATIVE_COUNTS</p>
/threads/time/ average	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where: locality#* is defining the <i>locality</i> for which the average time spent executing one <i>HPX</i>-thread</p>	<p>Returns the average time spent executing one <i>HPX</i>-thread on the given <i>locality</i> since application start. If the instance name is <code>total</code> the counter returns the average time spent executing one <i>HPX</i>-thread</p>	<p>None</p>
188	<p>locality#* is defining the <i>locality</i> for which the average time spent executing one <i>HPX</i>-thread</p>	<p>Chapter 2. What's so special about <i>HPX</i>? Returns the average time spent executing one <i>HPX</i>-thread for all worker threads (cores) on that <i>locality</i>. If the instance name is <code>worker-thread#*</code> the</p>	

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 <i>locality</i> id of the <i>locality</i> the number of components should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall number of currently active components of the specified type on the given <i>locality</i> .	The type of the component. This is the string which has been used while registering the component with <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_COMPONENT</i> .
/runtime/count/ action-invocation	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the number of action invocations should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall (local) invocation count of the specified action type on the given <i>locality</i> .	The action type. This is the string which has been used while registering the action with <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_ACTION</i> or <i>HPX_REGISTER_ACTION_ID</i> .
/runtime/count/ remote-action-invocation	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the number of action invocations should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall (remote) invocation count of the specified action type on the given <i>locality</i> .	The action type. This is the string which has been used while registering the action with <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_ACTION</i> or <i>HPX_REGISTER_ACTION_ID</i> .
/runtime/uptime	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the system uptime should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall time since application start on the given <i>locality</i> in nanoseconds.	None
/runtime/memory/ virtual	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the allocated virtual memory should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the amount of virtual memory currently allocated by the referenced <i>locality</i> (in bytes).	None
/runtime/memory/ resident	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the allocated resident memory should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the amount of resident memory currently allocated by the referenced <i>locality</i> (in bytes).	None
2.5. Manual			189
/runtime/memory/ total	locality#*/total where:	Returns the total available <i>locality</i> (in bytes)	None memory for use by the referenced

Table 2.35: Performance counters exposing PAPI hardware counters

Counter type	Counter instance formatting	Description	Parameters
<p>/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 <code>--hpx:list-counters</code> and <code>--papi-event-info=all</code> command line options.</p>	<p>locality#*/total or locality#*/worker-thread#* where: locality#* is defining the <i>locality</i> for which the current current accumulated value of all busy-loop counters of all worker threads should be queried. The <i>locality</i> id (given by *) is a (zero based) number identifying the <i>locality</i>. 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 <code>--hpx:threads</code>.</p>	<p>This counter returns the current count of occurrences of the specified PAPI event. This counter is available only if the configuration time constant <code>HPX_WITH_PAPI</code> is set to ON (default: OFF).</p>	None

Table 2.36: Performance counters for general statistics

Counter type	Counter instance formatting	Description	Parameters
/stat/perf/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.
/stat/perf/rollingaverage	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.
/stat/perf/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.
/stat/perf/rollingstddev	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.
2.5. Manual			191
/stat/perf/median	Any full performance counter name.	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 in-stance format-ting	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 wild-cards 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 wild-cards 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 wild-cards 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 wild-cards 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 wild-cards 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 wild-cards 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 wild-cards 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 wild-cards 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 wild-cards 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 wild-cards in the counter names will be expanded.

Note: The `/arithmetics` counters can consume an arbitrary number of other counters. For this reason those have to be specified as parameters (a comma separated list of counters appended after a '@'. For instance:

```
./bin/hello_world_distributed -t2 \
```

(continues on next page)

(continued from previous page)

```

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

```

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

```

./bin/hello_world_distributed -t2 \
--hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
--hpx:print-counter=/arithmetics/add@\
    /threads{locality#0/worker-thread#0}/count/cumulative,\
    /threads{locality#0/worker-thread#1}/count/cumulative

```

Table 2.38: Performance counters tracking parcel coalescing

Counter type	Counter instance formatting	Description	Parameters
/ coalescing/ count parcels	locality#*/ where: is the <i>locality</i> id of the <i>locality</i> the number of parcels for the given action should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	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 <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_ACTION</i> or <i>HPX_REGISTER_ACTION_ID</i> .
/ coalescing/ count messages	locality#*/ where: is the <i>locality</i> id of the <i>locality</i> the number of messages for the given action should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	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 <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_ACTION</i> or <i>HPX_REGISTER_ACTION_ID</i> .
/ coalescing/ count average parcels	locality#*/ where: is the <i>locality</i> id of the <i>locality</i> the number of messages for the given action should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	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 <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_ACTION</i> or <i>HPX_REGISTER_ACTION_ID</i> .
194 / coalescing/ time/ average arrival	locality#*/ where: is the <i>locality</i> id of the <i>locality</i> the number of messages for the given action should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	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 <i>HPX</i> , 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 [APEX²³⁸](#), 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 [CMake²³⁹](#) configuration. [TAU²⁴⁰](#) is an optional dependency when using [APEX²⁴¹](#).

To build *HPX* with [APEX²⁴²](#) add `HPX_WITH_APEX=ON`, and, optionally, `TAU_ROOT=$PATH_TO_TAU` to your [CMake²⁴³](#) configuration. In addition, you can override the tag used for [APEX²⁴⁴](#) with the `HPX_WITH_APEX_TAG` option. Please see the [APEX *HPX* documentation²⁴⁵](#) for detailed instructions on using [APEX²⁴⁶](#) 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:queuinglocal-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.

²³⁸ <https://khuck.github.io/xpress-apex/>

²³⁹ <https://www.cmake.org>

²⁴⁰ <https://www.cs.uoregon.edu/research/tau/home.php>

²⁴¹ <https://khuck.github.io/xpress-apex/>

²⁴² <https://khuck.github.io/xpress-apex/>

²⁴³ <https://www.cmake.org>

²⁴⁴ <https://khuck.github.io/xpress-apex/>

²⁴⁵ <https://khuck.github.io/xpress-apex/usage/#hpx-louisiana-state-university>

²⁴⁶ <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](https://en.wikipedia.org/wiki/Message_Passing_Interface)²⁴⁷ 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](https://en.wikipedia.org/wiki/Message_Passing_Interface)²⁴⁸ communication. [MPI](https://en.wikipedia.org/wiki/Message_Passing_Interface)²⁴⁹ tasks will then always run on the same thread, instead of potentially being stuck in a queue behind other threads.

Note that *HPX* thread pools are completely independent from each other in the sense that task stealing will never happen between different thread pools. However, tasks running on a particular thread pool can schedule tasks on another thread pool.

Note: It is simpler in some situations to schedule important tasks with high priority instead of using a separate thread pool.

Using the resource partitioner

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](https://en.wikipedia.org/wiki/Portable_Hardware_Locality) (HWLOC)²⁵⁰ and lets you iterate through the topology to add the wanted processing units to the thread pool. Be-

²⁴⁷ https://en.wikipedia.org/wiki/Message_Passing_Interface

²⁴⁸ https://en.wikipedia.org/wiki/Message_Passing_Interface

²⁴⁹ https://en.wikipedia.org/wiki/Message_Passing_Interface

²⁵⁰ <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 {
```

(continues on next page)

(continued from previous page)

```

    // 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.
```

(continues on next page)

(continued from previous page)

```

        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";
}

```

(continues on next page)

(continued from previous page)

```

        // 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 facilities.

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";

```

(continues on next page)

(continued from previous page)

```

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 checkpoints 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 and another function to remove it (or vice versa). Both `operator<<` and `operator>>` rely on a `.write()` and a `.read()` function respectively. In order to know how much data to read from the `std::istream`, the `operator<<` will write the size of the checkpoint before writing the checkpoint data. Correspondingly, the `operator>>` will read the size of the stored data before reading the data into 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 `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.
//
// [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>
```

(continues on next page)

(continued from previous page)

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

Additionally those applications need to link with the `iostreams` component. When using `cmake` this can be achieved by using the `COMPONENT_DEPENDENCIES` parameter, for instance:

```
include(HPX_AddExecutable)

add_hpx_executable(
    hello_world
    SOURCES hello_world.cpp
    COMPONENT_DEPENDENCIES iostreams
)
```

Note: The `hpx::cout` and `hpx::cerr` streams buffer all output locally until a `std::endl` or `std::flush` is encountered. That means that no output will appear on the console as long as either of those is explicitly used.

2.6 Additional material

- 2-day workshop held at CSCS in 2016
 - Recorded lectures²⁵¹
 - Slides²⁵²
- Tutorials repository²⁵³
- STEllAR Group blog posts²⁵⁴

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`²⁵⁵ can be used to generate a basic skeleton. The structure of this skeleton is as follows:

- *<lib_name>/*
 - *Readme.md*
 - *CMakeLists.txt*

²⁵¹ <https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOmIJ0lFNjKze18>

²⁵² <https://github.com/STELLAR-GROUP/tutorials/tree/master/cscs2016>

²⁵³ <https://github.com/STELLAR-GROUP/tutorials>

²⁵⁴ <http://stellar-group.org/blog/>

²⁵⁵ https://github.com/STELLAR-GROUP/hpx/blob/master/libs/create_library_skeleton.py

```

- cmake
- docs/ - index.rst
- examples/
    * CMakeLists.txt
- include/
    * hpx/
        · <lib_name>
- src/
    * CMakeLists.txt
- tests/
    * CMakeLists.txt
    * unit/
        · CMakeLists.txt
    * regressions/
        · CMakeLists.txt
    * performance/
        · CMakeLists.txt

```

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 separation. 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      =      de-
                                         fault_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<char>::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 is currently 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 is currently 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 : public error_code
```

`#include <error_code.hpp>` A `hpx::error_code` represents an arbitrary error condition.

The class `hpx::error_code` describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces.

Note Class `hpx::error_code` is an adjunct to error reporting by exception

Public Functions

```
error_code (throwmode mode = plain)
```

Construct an object of type `error_code`.

Parameters

- `mode`: The parameter `mode` specifies whether the constructed `hpx::error_code` belongs to the error category `hpx_category` (if `mode` is `plain`, this is the default) or to the category `hpx_category_rethrow` (if `mode` is `rethrow`).

Exceptions

- `nothing`:

```
error_code (error e, throwmode mode = plain)
```

Construct an object of type `error_code`.

Parameters

- `e`: The parameter `e` holds the `hpx::error` code the new exception should encapsulate.
- `mode`: The parameter `mode` specifies whether the constructed `hpx::error_code` belongs to the error category `hpx_category` (if `mode` is `plain`, this is the default) or to the category `hpx_category_rethrow` (if `mode` is `rethrow`).

Exceptions

- `nothing`:

```
error_code (error e, char const *func, char const *file, long line, throwmode mode = plain)
```

Construct an object of type `error_code`.

Parameters

- `e`: The parameter `e` holds the `hpx::error` code the new exception should encapsulate.
- `func`: The name of the function where the error was raised.
- `file`: The file name of the code where the error was raised.
- `line`: The line number of the code line where the error was raised.
- `mode`: The parameter `mode` specifies whether the constructed `hpx::error_code` belongs to the error category `hpx_category` (if `mode` is `plain`, this is the default) or to the category `hpx_category_rethrow` (if `mode` is `rethrow`).

Exceptions

- `nothing`:

error_code (*error* *e*, char **const** **msg*, *throwmode* *mode* = *plain*)

Construct an object of type *error_code*.

Parameters

- *e*: The parameter *e* holds the *hpx::error* code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- `std::bad_alloc`: (if allocation of a copy of the passed string fails).

error_code (*error* *e*, char **const** **msg*, char **const** **func*, char **const** **file*, long *line*, *throwmode* *mode* = *plain*)

Construct an object of type *error_code*.

Parameters

- *e*: The parameter *e* holds the *hpx::error* code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *func*: The name of the function where the error was raised.
- *file*: The file name of the code where the error was raised.
- *line*: The line number of the code line where the error was raised.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- `std::bad_alloc`: (if allocation of a copy of the passed string fails).

error_code (*error* *e*, std::string **const** &*msg*, *throwmode* *mode* = *plain*)

Construct an object of type *error_code*.

Parameters

- *e*: The parameter *e* holds the *hpx::error* code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- `std::bad_alloc`: (if allocation of a copy of the passed string fails).

error_code (*error* *e*, std::string **const** &*msg*, char **const** **func*, char **const** **file*, long *line*, *throwmode* *mode* = *plain*)

Construct an object of type *error_code*.

Parameters

- `e`: The parameter `e` holds the `hpx::error` code the new exception should encapsulate.
- `msg`: The parameter `msg` holds the error message the new exception should encapsulate.
- `func`: The name of the function where the error was raised.
- `file`: The file name of the code where the error was raised.
- `line`: The line number of the code line where the error was raised.
- `mode`: The parameter `mode` specifies whether the constructed `hpx::error_code` belongs to the error category `hpx_category` (if `mode` is `plain`, this is the default) or to the category `hpx_category_rethrow` (if `mode` is `rethrow`).

Exceptions

- `std::bad_alloc`: (if allocation of a copy of the passed string fails).

`std::string get_message () const`

Return a reference to the error message stored in the `hpx::error_code`.

Exceptions

- `nothing`:

`void clear ()`

Clear this `error_code` object. The postconditions of invoking this method are.

- `value() == hpx::success` and `category() == hpx::get_hpx_category()`

`error_code (error_code const &rhs)`

Copy constructor for `error_code`

Note This function maintains the error category of the left hand side if the right hand side is a success code.

`error_code &operator= (error_code const &rhs)`

Assignment operator for `error_code`

Note This function maintains the error category of the left hand side if the right hand side is a success code.

Private Functions

`error_code (int err, hpx::exception const &e)`

`error_code (std::exception_ptr const &e)`

Private Members

`std::exception_ptr exception_`

Friends

friend `hpx::error_code::exception`

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

class `exception` : **public** `system_error`

#include <exception.hpp> A `hpx::exception` is the main exception type used by HPX to report errors.

The `hpx::exception` type is the main exception type used by HPX to report errors. Any exceptions thrown by functions in the HPX library are either of this type or of a type derived from it. This implies that it is always safe to use this type only in catch statements guarding HPX library calls.

Subclassed by `hpx::exception_list`, `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: public hpx::exception

#include <exception_list.hpp> The class *exception_list* is a container of *exception_ptr* objects parallel algorithms may use to communicate uncaught exceptions encountered during parallel execution to the caller of the algorithm

The type *exception_list::const_iterator* fulfills the requirements of a forward iterator.

Public Types

typedef exception_list_type::const_iterator iterator
bidirectional iterator

Public Functions

std::size_t size() const

The number of *exception_ptr* objects contained within the *exception_list*.

Note Complexity: Constant time.

exception_list_type::const_iterator begin() const

An iterator referring to the first *exception_ptr* object contained within the *exception_list*.

exception_list_type::const_iterator end() const

An iterator which is the past-the-end value for the *exception_list*.

Private Types

```
typedef 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
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:
template<typename T>
struct is_async_execution_policy: public execution::detail::is\_async\_execution\_policy<hpx::util::decay<T>::type>
    #include <is_execution_policy.hpp> Extension: Detect whether given execution policy makes algorithms asyn-
    chronous
```

1. The type [is_async_execution_policy](#) can be used to detect asynchronous execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

2. If *T* is the type of a standard or implementation-defined execution policy, `is_async_execution_policy<T>` shall be publicly derived from `integral_constant<bool, true>`, otherwise from `integral_constant<bool, false>`.
3. The behavior of a program that adds specializations for *is_async_execution_policy* is undefined.

```
template<typename T>
struct is_execution_policy : public execution::detail::is_execution_policy<hpx::util::decay<T>::type>
#include <is_execution_policy.hpp>
```

1. The type *is_execution_policy* can be used to detect execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.
2. If *T* is the type of a standard or implementation-defined execution policy, `is_execution_policy<T>` shall be publicly derived from `integral_constant<bool, true>`, otherwise from `integral_constant<bool, false>`.
3. The behavior of a program that adds specializations for *is_execution_policy* is undefined.

```
template<typename T>
struct is_parallel_execution_policy : public execution::detail::is_parallel_execution_policy<hpx::util::decay<T>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy enables parallelization
```

1. The type *is_parallel_execution_policy* can be used to detect parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.
2. If *T* is the type of a standard or implementation-defined execution policy, `is_parallel_execution_policy<T>` shall be publicly derived from `integral_constant<bool, true>`, otherwise from `integral_constant<bool, false>`.
3. The behavior of a program that adds specializations for *is_parallel_execution_policy* is undefined.

```
template<typename T>
struct is_sequenced_execution_policy : public execution::detail::is_sequenced_execution_policy<hpx::util::decay<T>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy does not enable parallelization
```

1. The type *is_sequenced_execution_policy* can be used to detect non-parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.
2. If *T* is the type of a standard or implementation-defined execution policy, `is_sequenced_execution_policy<T>` shall be publicly derived from `integral_constant<bool, true>`, otherwise from `integral_constant<bool, false>`.
3. The behavior of a program that adds specializations for *is_sequenced_execution_policy* is undefined.

```
struct launch : public detail::policy_holder<>
#include <launch_policy.hpp> Launch policies for hpx::async etc.
```

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*...>::type

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 ¶meters ()

Return the associated executor parameters object.

executor_parameters_type const ¶meters () 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: **public** *hpx::parallel::execution::parallel_policy*

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

Public Types

typedef Executor **executor_type**

The type of the executor associated with this execution policy.

typedef Parameters **executor_parameters_type**

The type of the associated executor parameters object which is associated with this execution policy

typedef *hpx::traits::*executor_execution_category<*executor_type*>::type **execution_category**

The category of the execution agents created by this execution policy.

Public Functions

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 ¶meters()** **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...>::t
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: public hpx::parallel::execution::parallel_task_policy
```

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

Public Types

typedef Executor **executor_type**

The type of the executor associated with this execution policy.

typedef Parameters **executor_parameters_type**

The type of the associated executor parameters object which is associated with this execution policy

typedef *hpx::traits::*executor_execution_category<*executor_type*>::type **execution_category**

The category of the execution agents created by this execution policy.

Public Functions

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 **¶meters ()**
Return the associated executor parameters object.

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

executor_parameters_type **const ¶meters () 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*, *run-*
time_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* rp-
mode = *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 = *re-*
source::mode_default, *runtime_mode* mode = *runtime_mode_default*)

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

void **create_thread_pool** (std::string **const** &name, *scheduling_policy* sched = *schedul-*
ing_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 ex-
    clusive = 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 rebound
```

```
    #include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy
```

Public Types

```
typedef parallel_task_policy_shim<Executor_, Parameters_> type
```

```
    The type of the rebound execution policy.
```

```
template<typename Executor_, typename Parameters_>
```

struct 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 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

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 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 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_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_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...>::t
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: public hpx::parallel::execution::sequenced_policy
```

`#include <execution_policy.hpp>` The class `sequenced_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.

Public Types

```
typedef Executor executor_type
```

The type of the executor associated with this execution policy.

```
typedef Parameters executor_parameters_type
```

The type of the associated executor parameters object which is associated with this execution policy

```
typedef hpx::traits::executor_execution_category<executor_type>::type execution_category
```

The category of the execution agents created by this execution policy.

Public Functions

```
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 **¶meters** ()

Return the associated executor parameters object.

Parameters **const ¶meters** () **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...>::t
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**: **public** `hpx::parallel::execution::sequenced_task_policy`

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

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

Public Types

typedef `Executor` **executor_type**

The type of the executor associated with this execution policy.

typedef `Parameters` **executor_parameters_type**

The type of the associated executor parameters object which is associated with this execution policy

typedef `hpx::traits::executor_execution_category<executor_type>::type` **execution_category**

The category of the execution agents created by this execution policy.

Public Functions

`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 **¶meters** ()

Return the associated executor parameters object.

Parameters **const ¶meters** () **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: public [hpx::exception](#)

#include <task_block.hpp> The class [task_canceled_exception](#) defines the type of objects thrown by [task_block::run](#) or [task_block::wait](#) if they detect that an exception is pending within the current parallel region.

Public Functions

task_canceled_exception()

struct thread_interrupted: public [exception](#)

#include <exception.hpp> A [hpx::thread_interrupted](#) is the exception type used by HPX to interrupt a running HPX thread.

The [hpx::thread_interrupted](#) type is the exception type used by HPX to interrupt a running thread.

A running thread can be interrupted by invoking the `interrupt()` member function of the corresponding `hpx::thread` object. When the interrupted thread next executes one of the specified interruption points (or if it is currently blocked whilst executing one) with interruption enabled, then a [hpx::thread_interrupted](#) exception will be thrown in the interrupted thread. If not caught, this will cause the execution of the interrupted thread to terminate. As with any other exception, the stack will be unwound, and destructors for objects of automatic storage duration will be executed.

If a thread wishes to avoid being interrupted, it can create an instance of [hpx::this_thread::disable_interruption](#). Objects of this class disable interruption for the thread that created them on construction, and restore the interruption state to whatever it was before on destruction.

```
void f()
{
    // interruption enabled here
    {
        hpx::this_thread::disable_interruption di;
        // interruption disabled
    }
}
```

(continues on next page)

(continued from previous page)

```

    {
        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()`.

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

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

Parameters

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

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

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

Parameters

- `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* = 0)Suspend 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* = 0)Resumes the given processing unit. Takes a callback as a parameter which will be called when the processing unit has been resumed.

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

Parameters

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

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

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_default) Main entry point for launching the HPX runtime system.

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

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

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

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `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_default*) Main entry point for
launching the HPX runtime system.

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

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

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

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `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_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). 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_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). 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_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). 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_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). 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_nonsr<int>* int, char **

> **const** &*f*, 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). 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_nonsr<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_default Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

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 `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_default
```

Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

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 `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 `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_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. 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_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. 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_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. 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_noner<int> int, char **
> const &f, int argc, char **argv, 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. 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_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. 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 an 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_t(-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.

*namng::*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.

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 (comp
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, er
ror_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 an error.

Note The component instance the returned pointer refers to can not be migrated as long as there is at least one copy of the returned `shared_ptr` alive.

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

Parameters

- `p`: [in] The parameter `p` represents a placeholder type to turn make the call synchronous.
- `id`: [in] The global id of the component for which the pointer to the underlying memory should be retrieved.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

Template Parameters

- The: only template parameter has to be the type of the server side component.

```
template<typename Derived, typename Stub>
std::shared_ptr<typename components::client_base<Derived, Stub>::server_component_type> get_ptr (launch::sync_policy
                                                                                               p,
                                                                                               com-
                                                                                               po-
                                                                                               nents::client_base<Derived, Stub>
                                                                                               Stub>
                                                                                               const
                                                                                               &c,
                                                                                               er-
                                                                                               ror_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 parcel ports.

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.


```
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::id_type const &id, naming::address &&addr, Result &&t, bool  
    move_credits = true)  
    Set the result value for the (managed) LCO referenced by the given id.
```

Parameters

- *id*: [in] This represents the id of the LCO which should receive the given value.
- *t*: [in] This is the value which should be sent to the LCO.
- *move_credits*: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

```
template<typename Result>  
std::enable_if<!std::is_same<typename util::decay<Result>::type, naming::address>::value>::type set_lco_value_unmanaged (naming::id_type const &id, naming::address &&addr, Result &&t, bool  
    move_credits = true)  
    Set the result value for the (unmanaged) LCO referenced by the given id.
```

Parameters

- *id*: [in] This represents the id of the LCO which should receive the given value.
- *t*: [in] This is the value which should be sent to the LCO.
- *move_credits*: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.


```
template<typename Result>
void set_lco_value (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::id_type const &id, Result &&t, naming::id_type const &cont, bool move_credits = true)
```

Set the result value for the (managed) LCO referenced by the given id.

Parameters

- *id*: [in] This represents the id of the LCO which should receive the given value.
- *t*: [in] This is the value which should be sent to the LCO.
- *cont*: [in] This represents the LCO to trigger after completion.
- *move_credits*: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

```
template<typename Result>
```

```
std::enable_if<!std::is_same<typename util::decay<Result>::type, naming::address>::value>::type set_lco_value_unman
```

Set the result value for the (unmanaged) LCO referenced by the given id.

Parameters

- *id*: [in] This represents the id of the LCO which should receive the given value.
- *t*: [in] This is the value which should be sent to the LCO.
- *cont*: [in] This represents the LCO to trigger after completion.
- *move_credits*: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

```
void set_lco_error(naming::id_type const &id, naming::address &&addr, std::exception_ptr  
                 const &e, bool move_credits = true)
```

Set the error state for the LCO referenced by the given id.

Parameters

- *id*: [in] This represents the id of the LCO which should receive the error value.
- *addr*: [in] This represents the addr of the LCO which should be triggered.
- *e*: [in] This is the error value which should be sent to the LCO.
- *move_credits*: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

```
void set_lco_error(naming::id_type const &id, naming::address &&addr, std::exception_ptr  
                 &&e, bool move_credits = true)
```

Set the error state for the LCO referenced by the given id.

Parameters

- *id*: [in] This represents the id of the LCO which should receive the error value.
- *addr*: [in] This represents the addr of the LCO which should be triggered.
- *e*: [in] This is the error value which should be sent to the LCO.
- *move_credits*: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

```
void set_lco_error(naming::id_type const &id, std::exception_ptr const &e, bool move_credits  
                 = true)
```

Set the error state for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the error value.
- `e`: [in] This is the error value which should be sent to the LCO.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*naming::id_type* **const** &*id*, std::exception_ptr &&*e*, bool *move_credits* = true)
Set the error state for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the error value.
- `e`: [in] This is the error value which should be sent to the LCO.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*naming::id_type* **const** &*id*, *naming::address* &&*addr*, std::exception_ptr **const** &*e*, *naming::id_type* **const** &*cont*, bool *move_credits* = true)
Set the error state for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the error value.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `e`: [in] This is the error value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*naming::id_type* **const** &*id*, *naming::address* &&*addr*, std::exception_ptr &&*e*, *naming::id_type* **const** &*cont*, bool *move_credits* = true)
Set the error state for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the error value.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `e`: [in] This is the error value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*naming::id_type* **const** &*id*, std::exception_ptr **const** &*e*, *naming::id_type* **const** &*cont*, bool *move_credits* = true)
Set the error state for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the error value.
- `e`: [in] This is the error value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to `true` then it is ok to send all credits in `id` along with the generated message. The default value is `true`.

void **set_lco_error** (*naming*::id_type const &*id*, std::exception_ptr &&*e*, *naming*::id_type const &*cont*, bool *move_credits* = true)
Set the error state for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the error value.
- `e`: [in] This is the error value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to `true` then it is ok to send all credits in `id` along with the generated message. The default value is `true`.

template<typename *Component*, typename ... *Ts*><unspecified> **hpx::new_**(id_type const & *id*, ...)
Create one or more new instances of the given *Component* type on the specified locality.

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

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

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

Return The function returns different types depending on its use:

- If the explicit template argument *Component* represents a component type (`traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which can be used to retrieve the global address of the newly created component.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will return a new instance of that type which can be used to refer to the newly created component instance.

Parameters

- `locality`: [in] The global address of the locality where the new instance should be created on.
- `vs`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

template<typename *Component*, typename ... *Ts*><unspecified> **hpx::local_new**(*Ts* &&... *vs*)
Create one new instance of the given *Component* type on the current locality.

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

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

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

Return The function returns different types depending on its use:

- If the explicit template argument *Component* represents a component type (`traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which can be used to retrieve the global address of the newly created component. If the first argument is *hpx::launch::sync* the function will directly return an *hpx::id_type*.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will return a new instance of that type which can be used to refer to the newly created component instance.

Note The difference of this function to *hpx::new_* is that it can be used in cases where the supplied arguments are non-copyable and non-movable. All operations are guaranteed to be local only.

Parameters

- *vs*: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

template<typename Component, typename ... Ts>unspecified hpx::new_(id_type const & l
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::n

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

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

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

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

Return The function returns different types depending on its use:

- If the explicit template argument *Component* represents a component type (`traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which can be used to retrieve the global address of the newly created component.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will return a new instance of that type which can be used to refer to the newly created component instance.

Parameters

- `policy`: [in] The distribution policy used to decide where to place the newly created.
- `vs`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

template<typename Component, typename DistPolicy, typename ... Ts><unspecified> hpx::n

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

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

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

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

Return The function returns different types depending on its use:

- If the explicit template argument *Component* represents an array of a component type (i.e. *Component*[], where `traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which holds a `std::vector<hpx::id_type>`, where each of the items in this vector is a global address of one of the newly created components.
- If the explicit template argument *Component* represents an array of a client side object type (i.e. *Component*[], where `traits::is_client<Component>::value` evaluates to true), the

function will return an *hpx::future* object instance which holds a `std::vector<hpx::id_type>`, where each of the items in this vector is a client side instance of the given type, each representing one of the newly created components.

Parameters

- `policy`: [in] The distribution policy used to decide where to place the newly created.
- `count`: [in] The number of component instances to create
- `vs`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

```
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 hpx::wait_any(std::array< future< R >, N > & f
```

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

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

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

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

Parameters

- `futures`: [in] Amn array holding an arbitrary amount of *future* or *shared_future* objects for which *wait_any* should wait.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
template<typename ...T>
```

```
void wait_any(error_code &ec, T&&... futures)
```

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

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

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

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

Parameters

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

```
template<typename ...T>  
void wait_any(T&&... futures)
```

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

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

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

Parameters

- `futures`: [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_any* should wait.

```
template<typename InputIter>  
InputIter wait_any_n(InputIter first, std::size_t count, error_code &ec = throws)
```

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

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

Return The function *wait_all_n* will return an iterator referring to the first element in the input sequence after the last processed element.

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

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

Parameters

- `first`: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_any_n* should wait.
- `count`: [in] The number of elements in the sequence starting at *first*.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
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, Itera-
                                                                    tor first,
                                                                    Iterator
                                                                    last, er-
                                                                    ror_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, er-
```

```
ror_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(h
```

Perform a distributed broadcast operation.

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

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

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

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

Parameters

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

```
template<typename Action, typename ArgN, ...>void hpx::lcos::broadcast_apply(std::vector
```

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(h
```

Perform a distributed broadcast operation.

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

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

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

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

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

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.

- `argN`: [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

template<typename Action, typename ArgN, ...>void hpx::lcos::broadcast_apply_with_index

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

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

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<typename FoldOp::result_type>
    inverse_fold(const std::vector<hpx::id_type>& ids, FoldOp f, const Init& init, ArgN... args)
    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<typename FoldOp::result_type>
    inverse_fold_with_index(const std::vector<hpx::id_type>& ids, FoldOp f, const Init& init, ArgN... args)
    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 genera-
                                     tion = 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(A
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(A
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 generate the unmanaged global id from. This parameter can be itself a managed or a unmanaged global id.

`namespace parallel`

`namespace execution`

Typedefs

`using parallel_executor = parallel_policy_executor<hpx::launch>`

`using service_executor = threads::executors::service_executor`

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 io_pool_executor = threads::executors::io_pool_executor`

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 parcel_pool_executor = threads::executors::parcel_pool_executor`

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 timer_pool_executor = threads::executors::timer_pool_executor`

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 main_pool_executor = threads::executors::main_pool_executor`

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 local_priority_queue_executor = threads::executors::local_priority_queue_executor  
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 [anonymous]
```

```
namespace [anonymous]
```

```
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 copyExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 destCopies the ele-
ments 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_nExPolicy &&policy, FwdIter1 first, Size count, FwdIter2 destCopies the ele-
ments 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::projection_identity,
        typename 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
&&pol-
icy,
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
&&pol-
icy,
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 $\min(\text{last1} - \text{first1}, \text{last2} - \text{first2})$ 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 *i*th input element in the *i*th sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Return The *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 $1 < K+1 = M \leq N$.

```
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: $O(\text{last} - \text{first})$ 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 *i*th input element in the *i*th 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 $1 < K+1 = M \leq N$.

```
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, typename P =  
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 $S \cdot (N - S + 1)$ comparisons where $S = \text{distance}(\text{first2}, \text{last2})$ and $N = \text{distance}(\text{first1}, \text{last1})$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `FwdIter1`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *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, typename Pr
```

`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 $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj1**: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced *FwdIter1*.
- **Proj2**: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced *FwdIter2*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `s_first`: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- `s_last`: Refers to the end of the sequence of elements of the algorithm will be searching for.
- `op`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
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_identity,
        util::detail::algorithm_result<ExPolicy, FwdIter>::type for_each_n (ExPolicy &&policy, FwdIter first,
                                                                    Size count, F &&f, Proj &&proj =
                                                                    Proj())
```

Applies *f* to the result of dereferencing every iterator in the range `[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_identity
```

```
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 $N = \text{last} - \text{first}$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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* 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_identity>
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 $[\text{first}, \text{it})$ 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 $N = \text{last} - \text{first}$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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* 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 $2 \cdot (N1 + N2 - 1)$ comparisons, where $N1 = \text{std::distance}(\text{first1}, \text{last1})$ and $N2 = \text{std::distance}(\text{first2}, \text{last2})$.

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 $[\text{result}, \text{result} + (\text{last} - \text{first}))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM(op , init , $*\text{first}, \dots, *(\text{first} + (i - \text{result}))$).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 i th input element in the i th sum. If op is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Return The *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 , $\text{a1}, \dots, \text{aN}$) is defined as:

- a_1 when N is 1
- $op(\text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, a_1, \dots, a_K), \text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, a_M, \dots, a_N))$ where $1 < K+1 = M \leq N$.

```
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 $[\text{result}, \text{result} + (\text{last} - \text{first}))$ the value of $\text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, *first, \dots, *(first + (i - \text{result})))$.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 i th input element in the i th 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 $1 < K+1 = M \leq N$.

```
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: $O(\text{last} - \text{first})$ applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *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 $1 < K+1 = M \leq N$.

```
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 $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used for the 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 $N1 = \text{std::distance}(\text{first1}, \text{last})$ and $N2 = \text{std::distance}(\text{first2}, \text{last2})$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *Copy-Constructible*. This defaults to `std::less<>`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2**: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2**: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **pred**: Refers to the comparison function that the first and second ranges will be applied to

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Return The *lexicographically_compare* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

```
template<typename ExPolicy, typename RandIter1, typename RandIter2, typename RandIter3, typename Comp = d
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in1 (RandIter1) , tag::in2
```

```
RandIter2, tag::outRandIter3>>::type mergeExPolicy &&policy, RandIter1 first1, RandIter1 last1, Ran
dIter2 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 be-
ginning 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 $O(\text{std::distance}(\text{first1}, \text{last1}) + \text{std::distance}(\text{first2}, \text{last2}))$ 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_identity,
        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 $O(\text{std::distance}(\text{first}, \text{last}))$ applications of the comparison `comp` and the each projection.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `RandIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a 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::less>
util::detail::algorithm_result<ExPolicy, FwdIter>::type min_element (ExPolicy &&policy, FwdIter
                                                                    first, FwdIter last, F &&f = F(),
                                                                    Proj &&proj = Proj())
```

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std::distance}(\text{first}, \text{last})$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (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::less>
util::detail::algorithm_result<ExPolicy, FwdIter>::type max_element (ExPolicy &&policy, FwdIter
                                                                    first, FwdIter last, F &&f = F(),
                                                                    Proj &&proj = Proj())
```

Finds the greatest element in the range `[first, last)` using the given comparison function `f`.

The comparisons in the parallel `max_element` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std::distance}(first, last)$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `max_element` requires `F` to meet the requirements of `CopyConstructible`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the `This` argument is optional and defaults to `std::less`. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
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::less>
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::min (FwdIter) , tag::max
FwdIter>>::type minmax_elementExPolicy &&policy, FwdIter first, FwdIter last, F &&f = F(), Proj
&&proj = Proj()) Finds the greatest element in the range [first, last) using the given comparison function f.
```

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $\max(\text{floor}(3/2*(N-1)), 0)$ applications of the predicate, where $N = \text{std::distance}(\text{first}, \text{last})$.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (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<FwdIter1, FwdIter2>>>
```

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* mismatches `*(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<FwdIter1, FwdIter2>>>
```

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 moveExPolicy &&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 *BidItIter* 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, typename Proj,
        util::detail::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in (FwdIter1), tag::out1
```

```
FwdIter2, tag::out2FwdIter3>>::type partition_copyExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest_true, FwdIter3 dest_false, Pred &&pred, Proj &&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` 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 FwdIterB, typename FwdIterE, 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: $O(\text{last} - \text{first})$ applications of the predicate `f`.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIterB**: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIterE**: The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&. The types *Type1* *Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.

- **init**: The initial value for the generalized sum.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

Note GENERALIZED_SUM(op, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, ..., bK), GENERALIZED_SUM(op, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - 1 < K+1 = M <= N.

```
template<typename ExPolicy, typename FwdIterB, typename FwdIterE, 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: $O(\text{last} - \text{first})$ applications of the operator`+`().

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIterB`: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIterE`: The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to 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
 - $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIterB, typename FwdIterE>
```

`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: $O(\text{last} - \text{first})$ applications of the operator+().

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIterB**: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIterE**: The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a `hpx::future<T>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns T otherwise (where T is the value_type of *FwdIterB*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

Note The type of the initial value (and the result type) *T* is determined from the value_type of the used *FwdIterB*.

Note GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - $1 < K+1 = M \leq N$.

template<typename **ExPolicy**, typename **RanIter**, typename **RanIter2**, typename **FwdIter1**, typename **FwdIter2**, typ

```
util::detail::algorithm_result<ExPolicy, std::pair<FwdIter1, FwdIter2>>::type reduce_by_key (ExPolicy
&&pol-
icy,
Ran-
Iter
key_first,
Ran-
Iter
key_last,
Ran-
Iter2
val-
ues_first,
FwdIter1
keys_output,
FwdIter2
val-
ues_output,
Com-
pare
&&comp
=
Com-
pare(),
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: $O(\text{last} - \text{first})$ applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it 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::projection_identity>
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::in(FwdIter1), tag::out
```

```
FwdIter2>>::type remove_copyExPolicy &&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::projection_identity,
        typename 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_identity>
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_identity>
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 be 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 Proj =
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::in (FwdIter1), tag::out
```

```
FwdIter2>>::type replace_copyExPolicy &&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 Proj = 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 be 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_copyExPolicy &&policy, BidirIter first, BidirIter last, FwdIter
dest_firstCopies 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 rotateExPolicy &&policy, FwdIter first, FwdIter new_first, FwdIter lastPerforms 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, typename Pr
```

```
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 ($S \cdot N$) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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, typename Proj1 =
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 $S = \text{distance}(s_first, s_last)$ and $N = \text{count}$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an 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 = detail::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 $2 \cdot (N1 + N2 - 1)$ 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 = detail::
```

`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 = detail::
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 *Copy-Constructible*. This defaults to `std::less<>`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2**: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2**: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **op**: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
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 = detail::
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 $2 \cdot (N1 + N2 - 1)$ 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 = detail::
util::detail::algorithm_result<ExPolicy, RandomIt>::type sort (ExPolicy &&policy, RandomIt first, Ran-
domIt last, Compare &&comp = Com-
pare(), 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: $O(N \log(N))$, where $N = \text{std::distance}(first, last)$ comparisons.

comp has to induce a strict weak ordering on the values.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **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_keyExPolicy &&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: $O(N\log(N))$, where $N = \text{std::distance}(\text{first}, \text{last})$ comparisons.

comp has to induce a strict weak ordering on the values.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **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::projection_identity,
        util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::in (FwdIter1), tag::out
```

*FwdIter2>>::type transformExPolicy &&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, typename P
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in1 (FwdIter1), tag::in2
```

FwdIter2, tag::out*FwdIter3*>>::type *transformExPolicy* &&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, typename P,
        util::detail::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in1 (FwdIter1) , tag::in2
```

```
    FwdIter2, tag::outFwdIter3>>::type transformExPolicy &&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 $\min(\text{last2}-\text{first2}, \text{last1}-\text{first1})$ 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 Conv>
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: $O(\text{last} - \text{first})$ 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 $1 < K+1 = M \leq N$.

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, typename T>
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: $O(\text{last} - \text{first})$ applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **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 $1 < K+1 = M \leq N$.

The difference between *exclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the *i*th input element in the *i*th 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 GENERAL-
```

IZED_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: $O(\text{last} - \text{first})$ applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **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 $1 < K+1 = M \leq N$.

The difference between *exclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the *i*th input element in the *i*th 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: $O(\text{last} - \text{first})$ 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
 - $1 < K+1 = M \leq N$.

```
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: $O(\text{last} - \text{first})$ applications of the predicate *op2*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the first source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be used as return) values (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.

- `first1`: Refers to the beginning of the first sequence of elements the result will be calculated with.
- `last1`: Refers to the end of the first sequence of elements the algorithm will be applied to.
- `first2`: Refers to the beginning of the second sequence of elements the result will be calculated with.
- `init`: The initial value for the sum.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Reduce, typename Convert,
        util::detail::algorithm_result<ExPolicy, T>::type transform_reduce (ExPolicy &&policy, FwdIter1
                                                                    first1, FwdIter1 last1,
                                                                    FwdIter2 first2, T init, Re-
                                                                    duce &&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: $O(\text{last} - \text{first})$ 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
                                                                                      &&pol-
                                                                                      icy,
                                                                                      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>::type
```

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>::type
```

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    &&pol-
                                                                    istry, 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_nExPolicy &&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
                                                                                      &&pol-
                                                                                      icy,
                                                                                      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::projection_identity,
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 a 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, typename Proj =  
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 a 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 copyExPolicy &&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_identity>
```

```
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 a 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
&&pol-
icy,
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::proj,
        typename ResType = 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 $S \cdot (N - S + 1)$ comparisons where $S = \text{distance}(\text{begin}(\text{rng2}), \text{end}(\text{rng2}))$ and $N = \text{distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$.

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 Proj1 = util::
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_first_of (ExPolicy
&&pol-
icy,
Rng1
&&rng1
Rng2
&&rng2
Pred
&&op
=
Pred(),
Proj1
&&proj1
=
Proj1(),
Proj2
&&proj2
=
Proj2())
```

Searches the range *rng1* for any elements in the range *rng2*. Uses binary predicate *p* to compare elements

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most $(S*N)$ comparisons where $S = \text{distance}(\text{begin}(\text{rng2}), \text{end}(\text{rng2}))$ and $N = \text{distance}(\text{begin}(\text{rng1}), \text{end}(\text{rng1}))$.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1:** The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Rng2:** The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Pred:** The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to`.
- **Proj1:** The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements in *rng1*.
- **Proj2:** The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements in *rng2*.

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **rng1:** Refers to the first sequence of elements the algorithm will be applied to.
- **rng2:** Refers to the second sequence of elements the algorithm will be applied to.
- **op:** The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
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 *Iter* 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
&&pol-
icy,
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_identity>
```

```
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 a random access iterator.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **comp**: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* 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_identity>
```

```

util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type is_heap_until (ExPolicy
&&policy,
Rng
&&rng,
Comp
&&comp
=
Comp(),
Proj
&&proj
=
Proj())

```

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **comp**: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* 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::less, type
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in1 (typename
hpx::traits::range_iterator<Rng1>::type),
tag::in2
```

typename *hpx::traits::range_iterator<**Rng2**>::type*, tag::out**RandIter3**>::type **merge***ExPolicy* &&*policy*, *Rng1* &&*rng1*, *Rng2* &&*rng2*, *RandIter3* *dest*, *Comp* &&*comp* = *Comp*(), *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*() Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs $O(\text{std::distance}(\text{first1}, \text{last1}) + \text{std::distance}(\text{first2}, \text{last2}))$ 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::p
util::detail::algorithm_result<ExPolicy, RandIter>::type inplace_merge (ExPolicy &&policy, Rng
&&rng, RandIter middle,
Comp &&comp = Comp(),
Proj &&proj = Proj())
```

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs $O(\text{std::distance}(\text{first}, \text{last}))$ applications of the comparison `comp` and the each projection.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 (ExPolicy &&pol
&&pol
icy,
Rng
&&rng
F
&&f
=
F(),
Proj
&&pro
=
Proj())
```

Finds the smallest element in the range `[first, last)` using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std::distance}(\text{first}, \text{last})$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 (ExPolicy
&&policy,
Rng
&&rng,
F
&&f
=
F(),
Proj
&&proj
=
Proj())
```

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *max*(N-1, 0) comparisons, where N = `std::distance(first, last)`.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *max_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.

- `f`: The binary predicate which returns true if the 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 $\max(\text{floor}(3/2 * (N-1)), 0)$ applications of the predicate, where $N = \text{std::distance}(\text{first}, \text{last})$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `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
&&pol-
icy,
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, typename Proj,
        typename Util = hpx::util::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in (typename
        hpx::traits::range_iterator<Rng>::type),
        tag::out1
```

`FwdIter2`, tag::out2**FwdIter3**>>::type **partition_copy**`ExPolicy &&policy, Rng &&rng, FwdIter2 dest_true, FwdIter3 dest_false, Pred &&pred, Proj &&proj = Proj()`Copies the elements in the range `rng`, to two different ranges depending on the value returned by the predicate `pred`. The elements, that satisfy the predicate `pred`, are copied to the range beginning at `dest_true`. The rest of the elements are copied to the range beginning at `dest_false`. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than N assignments, exactly N applications of the predicate `pred`, where `N = std::distance(begin(rng), end(rng))`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range for the elements that satisfy the predicate `pred` (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter3`: The type of the iterator representing the destination range for the elements that don't satisfy the predicate `pred` (deduced). This iterator type must meet the requirements of an forward iterator.

- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `partition_copy` requires `Pred` to meet the requirements of `CopyConstructible`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest_true`: Refers to the beginning of the destination range for the elements that satisfy the predicate `pred`.
- `dest_false`: Refers to the beginning of the destination range for the elements that don't satisfy the predicate `pred`.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range `rng`. This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `is` invoked.

The assignments in the parallel `partition_copy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `partition_copy` algorithm returns a `hpx::future<tagged_tuple<tag::in(InIter), tag::out1(OutIter1), tag::out2(OutIter2)>>` if the execution policy is of type `parallel_task_policy` and returns `tagged_tuple<tag::in(InIter), tag::out1(OutIter1), tag::out2(OutIter2)>` otherwise. The `partition_copy` algorithm returns the tuple of the source iterator `last`, the destination iterator to the end of the `dest_true` range, and the destination iterator to the end of the `dest_false` range.

```
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
&&pol-
icy,
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
&&pol-
icy,
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_identity>
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::in (typename
                                                                    hpx::traits::range_traits<Rng>::iterator_type),
                                                                    tag::out
```

OutIter>>::type remove_copyExPolicy &&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_identity>
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_ifExPolicy &&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 a 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_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type replace (ExPolicy
&&pol-
icy,
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
&&pol-
icy,
Rng
&&rng,
F
&&f,
T
const
&new_v
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 be 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::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_copyExPolicy &&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 = util::projection_identity,
        typename Tag = 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_ifExPolicy &&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 be 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
                                                                    &&pol-
                                                                    icy,
                                                                    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 rotateExPolicy &&policy, Rng &&rng,
typename hpx::traits::range_iterator<Rng>::type middlePerforms 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, OutIter new_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 Proj1 = util::
```

```

util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type search (ExPolicy
&&pol-
icy,
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 ($S \cdot N$) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 Proj1 = util::
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type search_n (ExPolicy
&&pol-
icy,
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 $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which

the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `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 = detail::less>
```

```

util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type sort (ExPolicy
&&pol-
icy,
Rng
&&rng,
Com-
pare
&&comp
=
Com-
pare(),
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: $O(N\log(N))$, where $N = \text{std::distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$ 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_identity>
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::in (typename
                                     hpx::traits::range_iterator<Rng>::type),
                                     tag::out
```

```
OutIter>>::type transformExPolicy &&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, begin-
ning 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 $\text{size}(rng)$ 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 Proj1 = ut
```

```
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in1 (typename
                                                                    hpx::traits::range_iterator<Rng>::type),
                                                                    tag::in2
```

InIter2, tag::out*OutIter*>>::type **transform***ExPolicy* &&*policy*, *Rng* &&*rng*, *InIter2* *first2*, *OutIter* *dest*, *F* &&*f*, *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*() Applies the given function *f* to pairs of elements from two ranges: one defined by *rng* and the other beginning at *first2*, and stores the result in another range, beginning at *dest*.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *size(rng)* applications of *f*

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *InIter2*: The type of the source iterators for the second range used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- *Proj1*: The type of an optional projection function to be used for elements of the first sequence. This defaults to *util::projection_identity*
- *Proj2*: The type of an optional projection function to be used for elements of the second sequence. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *first2*: Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [*first*, *last*). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have *const&*. The types *Type1* and *Type2* must be such that objects of types *InIter1* and *InIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *OutIter* can be dereferenced and assigned a value of type *Ret*.

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *f* is invoked.

- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate f is invoked.

The invocations of f in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *transform* algorithm returns a `hpx::future<tagged_tuple<tag::in1(InIter1), tag::in2(InIter2), tag::out(OutIter)>>` if the execution policy is of type *parallel_task_policy* and returns `tagged_tuple<tag::in1(InIter1), tag::in2(InIter2), tag::out(OutIter)>` otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename OutIter, typename F, typename Proj1 = util::  
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 transformExPolicy  
&&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 $\min(\text{last2}-\text{first2}, \text{last1}-\text{first1})$ 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_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type unique (ExPolicy
&&pol-
icy,
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 a 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 Proj = util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::in (typename
```

```
hpx::traits::range_iterator<Rng>::type), tag::out
```

```
FwdIter2>>::type unique_copyExPolicy &&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 a 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  
                                                                                   &&pol-  
                                                                                   icy,      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, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `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, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *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<I>::value || std::is_iterator<I>::value>

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
```

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_int
```

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
```

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, &
```

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<

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 $i + p * \text{stride}$ 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 $i + n * \text{stride}$ 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 $\text{var} = \text{combiner}(\text{var}, \text{var})$ 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 = throws) Install a new generic performance counter type in a way, which will uninstall it automatically
during shutdown.
```

The function `install_counter_type` will register a new generic counter type based on the provided function. The counter type will be automatically unregistered during system shutdown. Any consumer querying any instance of this 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 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 scheduler_function = util::function_nonser<std::unique_ptr<hpx::threads::thread_pool_base> (hpx::threads::pool_base::size_t,
                                                                                               std::size_t,
                                                                                               std::size_t,
                                                                                               std::string
                                                                                               const&) >
```

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 and 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_nosser*<bool> thread_id_type
> const &*f*, *thread_state_enum* state = *unknown*) The function *enumerate_threads* will invoke the given function *f* for each thread with a matching thread state.

Parameters

- *f*: [in] The function which should be called for each matching thread. Returning ‘false’ from this function will stop the enumeration process.
- state: [in] This specifies the thread-state for which the threads should be enumerated.

thread_state **set_thread_state** (thread_id_type const &*id*, *thread_state_enum* state = *pending*, *thread_state_ex_enum* stateex = *wait_signaled*, *thread_priority*
priority = *thread_priority_normal*, bool *retry_on_active* = true,
hpx::error_code &*ec* = *throws*)

Set the thread state of the *thread* referenced by the thread_id *id*.

Note If the thread referenced by the parameter *id* is in *thread_state::active* state this function schedules a new thread which will set the state of the thread as soon as its not active anymore. The function returns *thread_state::active* in this case.

Return This function returns the previous state of the thread referenced by the *id* parameter. It will return one of the values as defined by the *thread_state* enumeration. If the thread is not known to the thread-manager the return value will be *thread_state::unknown*.

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

Parameters

- *id*: [in] The thread id of the thread the state should be modified for.
- state: [in] The new state to be set for the thread referenced by the *id* parameter.
- stateex: [in] The new extended state to be set for the thread referenced by the *id* parameter.
- priority:
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

thread_id_type **set_thread_state** (thread_id_type const &*id*, *util::steady_time_point* const
&*abs_time*, std::atomic<bool> **started*, *thread_state_enum*
state = *pending*, *thread_state_ex_enum* stateex = *wait_timeout*,
thread_priority priority = *thread_priority_normal*, bool
retry_on_active = true, *error_code* &*ec* = *throws*)

Set the thread state of the *thread* referenced by the thread_id *id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (at the given time)

Return

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

Parameters

- *id*: [in] The thread id of the thread the state should be modified for.
- *abs_time*: [in] Absolute point in time for the new thread to be run
- *started*: [in,out] A helper variable allowing to track the state of the timer helper thread

- `state`: [in] The new state to be set for the thread referenced by the `id` parameter.
- `stateex`: [in] The new extended state to be set for the thread referenced by the `id` parameter.
- `priority`:
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
thread_id_type set_thread_state (thread_id_type const &id, util::steady_time_point
                                const &abs_time, thread_state_enum state = pending,
                                thread_state_ex_enum stateex = wait_timeout, thread_priority
                                priority = thread_priority_normal, bool retry_on_active = true,
                                error_code& = throws)
```

```
thread_id_type set_thread_state (thread_id_type const &id, util::steady_duration
                                const &rel_time, thread_state_enum state = pending,
                                thread_state_ex_enum stateex = wait_timeout, thread_priority
                                priority = thread_priority_normal, bool retry_on_active = true,
                                error_code &ec = throws)
```

Set the thread state of the *thread* referenced by the thread_id *id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (after the given duration)

Return

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

Parameters

- `id`: [in] The thread id of the thread the state should be modified for.
- `rel_time`: [in] Time duration after which the new thread should be run
- `state`: [in] The new state to be set for the thread referenced by the `id` parameter.
- `stateex`: [in] The new extended state to be set for the thread referenced by the `id` parameter.
- `priority`:
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
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`

can be used to disable all other options.

As the name suggests, this option

`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`

os-thread driving the scheduler will be reduced below normal.

The kernel priority of the

`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.

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.

`enable_stealing = 0x20`

schedulers to explicitly disable thread stealing

This option allows for certain

enable_idle_backoff = 0x40

schedulers to explicitly disable exponential idle-back off

This option allows for certain

default_mode = *do_background_work* | *reduce_thread_priority* | *delay_exit* | *enable_stealing* | *enable_idle_backoff*

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>::value && !hpx::future<checkpoint> save_checkpoint (T &&t, Ts&&... ts)`

Save_checkpoint

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. 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>::type,
checkpoint> save_checkpoint (hpx::launch::sync_policy sync_p, T &&t, Ts&&... ts)
Save_checkpoint - Sync_policy overload
```

`Save_checkpoint` takes any number of objects which a user may wish to store and returns a future to a checkpoint object. 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
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**
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**>
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_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 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)
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>
```

(continues on next page)

(continued from previous page)

```

    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 alternative 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 alternative 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 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          <cstdint>#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
<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>#include
<hpx/runtime/serialization/detail/non_default_constructible.hpp>#include <hpx/traits/needs_automatic_registration.hpp>#include
<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/parallel/executors/fused_bulk_execute.hpp>#include          <hpx/util/deferred_call.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/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/local/futures_factory.hpp>#include
<hpx/util/cache_aligned_data.hpp>#include
<hpx/traits/is_iterator.hpp>#include
<hpx/parallel/algorithms/detail/is_negative.hpp>#include
<hpx/parallel/executors/post_policy_dispatch.hpp>#include
<hpx/runtime/serialization/serialize.hpp>#include
<hpx/runtime/serialization/brace_initializable_fwd.hpp>#include
<hpx/runtime/serialization/input_archive.hpp>#include
<iostream>#include
<map>#include
<hpx/runtime/serialization/detail/pointer.hpp>#include
<hpx/runtime/serialization/detail/polymorphic_intrusive_factory.hpp>#include
<hpx/runtime/serialization/string.hpp>#include
<hpx/runtime/serialization/binary_filter.hpp>#include
<hpx/runtime/naming/name.hpp>#include
<hpx/util/fibhash.hpp>#include
<boost/version.hpp>#include
<hpx/traits/is_bitwise_serializable.hpp>#include
<hpx/runtime/naming/id_type_impl.hpp>#include
<cstring>#include
<hpx/traits/serialization_access_data.hpp>#include
<boost/predef/other/endian.h>#include
<hpx/runtime/serialization/output_container.hpp>#include
<hpx/traits/is_executor_parameters.hpp>#include
<hpx/util/bind_back.hpp>#include
<hpx/util/pack_traversal.hpp>#include
<hpx/lcos/wait_all.hpp>#include
<hpx/traits/detail/unwrap_int.hpp>#include
<stdexcept>#include
<hpx/components/component_storage/server/migrate_from_storage.hpp>#include
<hpx/runtime/components/runtime_support.hpp>#include
<hpx/runtime/agas_fwd.hpp>#include
<hpx/runtime/components/component_type.hpp>#include
<hpx/util/detail/pp/expand.hpp>#include
<hpx/runtime/naming/address.hpp>#include
<hpx/runtime/actions_fwd.hpp>#include
<hpx/util/thread_specific_ptr.hpp>#include
<hpx/async.hpp>#include
<hpx/lcos/detail/async_implementations_fwd.hpp>#include
<hpx/lcos/promise.hpp>#include
<hpx/lcos/detail/promise_lco.hpp>#include
<hpx/lcos/base_lco.hpp>#include
<hpx/lcos/sync_fwd.hpp>#include
<hpx/runtime/components/pinned_ptr.hpp>#include
<hpx/runtime/components_fwd.hpp>#include
<hpx/traits/action_decorate_function.hpp>#include
<hpx/runtime/serialization/base_object.hpp>#include
<hpx/traits/action_remote_result.hpp>#include
<hpx/runtime/actions/continuation.hpp>#include
<hpx/traits/action_priority.hpp>#include
<hpx/runtime/agas/interface.hpp>#include
<hpx/runtime/trigger_lco.hpp>#include
<hpx/traits/is_continuation.hpp>#include
<hpx/runtime/actions/detail/invoke_count_registry.hpp>#include
<hpx/runtime/actions/preassigned_action_id.hpp>#include
<hpx/runtime/actions/transfer_base_action.hpp>#include
<hpx/runtime/threads/thread_id_type.hpp>#include
<hpx/traits/action_does_termination_detection.hpp>#include
<hpx/lcos/async_fwd.hpp>#include
<hpx/lcos/local/latch.hpp>#include
<hpx/parallel/algorithms/detail/predicates.hpp>#include
<boost/iterator/iterator_categories.hpp>#include
<cstdlib>#include
<hpx/parallel/executors/static_chunk_size.hpp>#include
<hpx/runtime/serialization/access.hpp>#include
<hpx/traits/brace_initializable_traits.hpp>#include
<hpx/runtime/serialization/basic_archive.hpp>#include
<hpx/runtime/serialization/detail/raw_ptr.hpp>#include
<hpx/runtime/serialization/detail/polymorphic_id_factory.hpp>#include
<hpx/runtime/serialization/input_container.hpp>#include
<hpx/runtime/serialization/container.hpp>#include
<hpx/util/spinlock_pool.hpp>#include
<hpx/runtime/naming/id_type.hpp>#include
<hpx/util/detail/yield_k.hpp>#include
<hpx/runtime/serialization/serialization_chunk.hpp>#include
<boost/cstdint.hpp>#include
<hpx/runtime/serialization/output_archive.hpp>#include
<hpx/runtime/serialization/detail/polymorphic_nonintrusive_factory.hpp>#include
<hpx/parallel/executors/execution_parameters_fwd.hpp>#include
<hpx/util/detail/unwrap_impl.hpp>#include
<hpx/util/detail/pack_traversal_impl.hpp>#include
<stdexcept>#include
<hpx/runtime/applier/applier.hpp>#include
<hpx/runtime/applier_fwd.hpp>#include
<hpx/traits/component_type_database.hpp>#include
<hpx/util/detail/pp/nargs.hpp>#include
<hpx/util_fwd.hpp>#include
<hpx/runtime/parcelset/parcel.hpp>#include
<hpx/runtime/parcelset_fwd.hpp>#include
<hpx/runtime/components/stubs/runtime_support.hpp>#include
<hpx/lcos/detail/async_implementations.hpp>#include
<hpx/lcos/packaged_action.hpp>#include
<hpx/lcos/detail/promise_base.hpp>#include
<hpx/lcos/base_lco_with_value.hpp>#include
<hpx/runtime/actions/basic_action.hpp>#include
<hpx/runtime/actions/action_support.hpp>#include
<hpx/runtime/get_lva.hpp>#include
<hpx/traits/managed_component_policies.hpp>#include
<hpx/traits/component_pin_support.hpp>#include
<hpx/runtime/threads/thread_init_data.hpp>#include
<hpx/runtime/actions/basic_action_fwd.hpp>#include
<hpx/runtime/actions/action_priority.hpp>#include
<hpx/runtime/actions/trigger.hpp>#include
<boost/dynamic_bitset.hpp>#include
<hpx/runtime/applier/detail/apply_implementations_fwd.hpp>#include
<hpx/runtime/actions/detail/action_factory.hpp>#include
<hpx/performance_counters/counters_fwd.hpp>#include
<hpx/runtime/actions/transfer_action.hpp>#include
<hpx/runtime/actions/base_action.hpp>#include
<hpx/runtime/serialization/unique_ptr.hpp>#include
<hpx/traits/action_message_handler.hpp>#include

```

```

<hpx/traits/action_schedule_thread.hpp>#include
<hpx/traits/action_stacksize.hpp>#include
<hpx/util/get_and_reset_value.hpp>#include
<hpx/runtime_fwd.hpp>#include
<hpx/components_fwd.hpp>#include
<hpx/traits/is_client.hpp>#include
<hpx/runtime/find_localities.hpp>#include
<hpx/runtime/get_locality_id.hpp>#include
<hpx/runtime/get_num_localities.hpp>#include
<hpx/runtime/report_error.hpp>#include
<hpx/runtime/runtime_mode.hpp>#include
<hpx/runtime/shutdown_function.hpp>#include
<hpx/state.hpp>#include
<hpx/traits/action_continuation.hpp>#include
<hpx/traits/action_select_direct_execution.hpp>#include
<hpx/runtime/actions/transfer_continuation_action.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/component_heap.hpp>#include
<hpx/util/bind_front.hpp>#include
<hpx/util/static_reinit.hpp>#include
<hpx/util/generate_unique_ids.hpp>#include
<new>#include
<hpx/util/one_size_heap_list.hpp>#include
<hpx/plugins/parcel/coalescing_message_handler_registration.hpp>#include
<hpx/util/ini.hpp>#include
<boost/lexical_cast.hpp>#include
<boost/utility/swap.hpp>#include
<hpx/runtime/applier/detail/apply_implementations.hpp>#include
<hpx/traits/component_supports_migration.hpp>#include
<cc-type>#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/put_parcel.hpp>#include
<hpx/performance_counters/counters.hpp>#include
<hpx/runtime/serialization/map.hpp>#include
<hpx/util/runtime_configuration.hpp>#include
<hpx/util/plugin/export_plugin.hpp>#include
<hpx/util/plugin/virtual_constructor.hpp>#include
<boost/any.hpp>#include
<boost/shared_ptr.hpp>#include
<hpx/util/plugin/plugin_wrapper.hpp>#include
<hpx/util/plugin/dll.hpp>#include
<boost/filesystem/convenience.hpp>#include
<boost/filesystem/path.hpp>#include
<link.h>#include
<dlfcn.h>#include
<limits.h>#include
<hpx/util/plugin.hpp>#include
<boost/filesystem.hpp>#include
<set>#include
<hpx/runtime/parcelset/parcelhandler.hpp>#include
<hpx/performance_counters/parcels/data_point.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.hpp>#include
<hpx/runtime/serialization/array.hpp>#include
<boost/array.hpp>#include
<hpx/runtime/serialization/serialize_buffer.hpp>#include
<hpx/traits/supports_streaming_with_any>#include
<hpx/traits/action_serialization_filter.hpp>#include
<hpx/traits/action_was_object_migrated.hpp>#include
<hpx/runtime/applier/apply_helper.hpp>#include
<hpx/runtime/basename_registration_fwd.hpp>#include
<hpx/runtime/components/make_client.hpp>#include
<hpx/runtime/config_entry.hpp>#include
<hpx/runtime/get_colocation_id.hpp>#include
<hpx/runtime/get_locality_name.hpp>#include
<hpx/runtime/get_thread_name.hpp>#include
<hpx/runtime/runtime_fwd.hpp>#include
<hpx/runtime/set_parcel_write_handler.hpp>#include
<hpx/runtime/startup_function.hpp>#include
<hpx/traits/action_decorate_continuation.hpp>#include
<hpx/runtime/parcelset/detail/per_action_data_counter_registry.hpp>#include
<hpx/traits/is_distribution_policy.hpp>#include
<hpx/runtime/actions/component_action.hpp>#include
<hpx/runtime/components/server/create_component>#include
<hpx/util/reinitializable_static.hpp>#include
<hpx/runtime/components/server/wrapper_heap.hpp>#include
<hpx/util/wrapper_heap_base.hpp>#include
<hpx/runtime/components/server/wrapper_heap_list.hpp>#include
<hpx/util/unlock_guard.hpp>#include
<hpx/runtime/components/server/component_base>#include
<hpx/lcos/local/promise.hpp>#include
<hpx/runtime/applier/apply.hpp>#include
<hpx/traits/action_is_target_valid.hpp>#include
<hpx/util/format.hpp>#include
<hpx/runtime/components/client_base.hpp>#include
<hpx/lcos/detail/async_colocated_fwd.hpp>#include
<hpx/runtime/parcelset/detail/parcel_await.hpp>#include
<hpx/runtime.hpp>#include
<hpx/runtime/parcelset/locality.hpp>#include
<hpx/runtime/thread_hooks.hpp>#include
<hpx/runtime/components/static_factory_data.hpp>#include
<hpx/util/plugin/abstract_factory.hpp>#include
<hpx/util/plugin/config.hpp>#include
<hpx/util/plugin/concrete_factory.hpp>#include
<boost/algorithm/string/case_conv.hpp>#include
<hpx/util/plugin/detail/dll_dlopen.hpp>#include
<hpx/plugins/plugin_registry_base.hpp>#include
<hpx/util/plugin/plugin_factory.hpp>#include
<hpx/runtime/naming/split_gid.hpp>#include
<hpx/runtime/parcelset/parcelport.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.hpp>#include
<hpx/runtime/serialization/array.hpp>#include
<boost/array.hpp>#include
<hpx/runtime/serialization/serialize_buffer.hpp>#include
<hpx/traits/supports_streaming_with_any>#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_c
<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/traits/component_heap_type.hpp>#include <hpx/runtime/serialization/unordered_map.hpp>#include
<hpx/components/containers/container_distribution_policy.hpp>#include <hpx/components/containers/unordered/partition_uno
<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/detail/coroutine_self.hpp>#include
<tuple>#include <hpx/components/containers/unordered/unordered_map_segmented_iterator.hpp>#include
<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 <ar-
ray>#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 <cstdint>#include <cstdint>#include <ex-
    ception>#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 <util-
    ity>#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 <excep-
    tion>#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 <cstd-
    def>#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/assert.hpp>#include      <assert.h>#include
<cstdlib>#include      <iostream>#include      <hpx/util/detail/empty_function.hpp>#include
<hpx/util/detail/vtable/vtable.hpp>#include      <hpx/util/detail/vtable/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/detail/vtable/copyable_vtable.hpp>#include <new>#include <hpx/util/detail/vtable/serializable_function_vtable.hpp>#include
<hpx/runtime/serialization/detail/polymorphic_intrusive_factory.hpp>#include
<hpx/util/debug/demangle_helper.hpp>#include <typeinfo>#include <hpx/util/jenkins_hash.hpp>#include
<random>#include      <unordered_map>#include      <hpx/util/detail/function_registration.hpp>#include
<hpx/util/detail/vtable/serializable_vtable.hpp>#include      <hpx/util_fwd.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
<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 <cstdlib>#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      <cstdlib>
def>#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 <cstdint>#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/min.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>#include <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/uid_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>#include
<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/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/runtime/agas/primary_namespace.hpp>#include
<hpx/runtime/agas/server/primary_namespace.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 <cstdlib>#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 <cstdlib>#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 <cstdlib>#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/assert.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 <cstdlib>#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 <cstdlib>#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/traits/projected_range.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 <cstdint>#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 <cstdint>#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
<cstdint>#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 <cstdint>#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
<cstdint>#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 <cstdint>#include <cs-  
t-dint>#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 <cstdint>#include <csdtint>#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  
<cstdint>#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 <cstdint>#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 <cstdint>#include <itera-  
tor>#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 <al-
gorithm>#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 <nu-
meric>#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 <cstd-
def>#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 <vec-
tor>#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/accumulate.hpp>#include
<functional>#include <hpx/parallel/algorithms/detail/dispatch.hpp>#include
<hpx/parallel/algorithms/detail/distance.hpp>#include <iterator>#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 <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/identity.hpp>#include <hpx/util/iterator_adaptor.hpp>#include <hpx/util/lazy_conditional.hpp>#include
<hpx/util/result_of.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
<cstdlib>#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 <cstdlib-
def>#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 <algo-
rithm>#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 <algo-
rithm>#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 <algo-
rithm>#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 <algo-
rithm>#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/executors/execution_information.hpp>#include <hpx/parallel/executors/execution_parameters.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 <cstdlib>#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 <cstdint>#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 <cstdint>#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 <cstdint>#include <iterator>#include <numeric>#include <type_traits>#include <util-  
ity>#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/sync_launch_policy_dispatch.hpp>#include <hpx/lcos/sync_fwd.hpp>#include
<hpx/lcos/local/futures_factory.hpp>#include <hpx/runtime/launch_policy.hpp>#include
<hpx/traits/is_action.hpp>#include <hpx/util/invoke.hpp>#include <hpx/util/deferred_call.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 <al-
    gorithm>#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 <cstdint>#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 <std-
    def>#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 <cstdint>#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/local/latch.hpp>#include <hpx/parallel/algorithms/detail/predicates.hpp>#include
    <hpx/parallel/executors/fused_bulk_execute.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_back.hpp>#include
    <hpx/util/deferred_call.hpp>#include <hpx/util/internal_allocator.hpp>#include
    <hpx/util/invoke.hpp>#include <hpx/util/one_shot.hpp>#include <hpx/util/range.hpp>#include
    <hpx/util/unwrap.hpp>#include <algorithm>#include <cstdint>#include <type_traits>#include <util-
    ity>#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 <cstdint>#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/sync_launch_policy_dispatch.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 <cstdint>#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 <cstdint>#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
<cstdint>#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
<cstdint>#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>#include
<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 <mem-
ory>#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 <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/invoke_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 <cstdint>#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>
    {
    public:
        void print_greeting ()
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }

        // Component actions need to be declared, this also defines the
```

(continues on next page)

(continued from previous page)

```

        // 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 <cs-
t-dlib>#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>
    {
    public:
        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 <cs-
t-dlib>#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 <cstdlib>#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 <cstdlib>#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>#include
<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 <cstdlib>#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 <al-
gorithm>#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/launch_policy.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 <cs-
    dint>

file get_os_thread_count.hpp
    #include <hpx/config.hpp>#include <hpx/runtime/threads/thread_data_fwd.hpp>#include <cstdint>

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 <cstdint>

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 <cstdint>#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 <cstdint>#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 <cstdint>#include <mem-
    ory>#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>#include
<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
<cstdlib>#include <cstdint>#include <utility>#include <memory>
```

file thread_enums.hpp

```
#include <hpx/config.hpp>#include <hpx/runtime/threads/detail/combined_tagged_state.hpp>#include <cstdlib>
#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 <cstdlib>#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 <cstdlib>#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 <ios-
fwd>#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 <cstdlib>#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/locals/future.hpp>#include <hpx/runtime/serialization/serialize.hpp>#include
<hpx/runtime/serialization/vector.hpp>#include <cstdint>#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 <cstd-
    def>#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. This step does not apply to patch releases. For both APEX and hpxMP:
 - Change the release branch to be the most current release tag available in the APEX/hpxMP `git_external` section in the main `CMakeLists.txt`. Please contact the maintainers of the respective packages to generate a new release to synchronize with the HPX release (APEX²⁵⁶, hpxMP²⁵⁷).
7. 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.
8. Unprotect the release branch in the github repository settings so that it can be deleted and recreated.
9. 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`

²⁵⁶ <http://github.com/khuck/xpress-apex>

²⁵⁷ <https://github.com/STELLAR-GROUP/hpxMP>

- `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`

10. Protect the release branch again to disable deleting and force pushes.
11. Check out the release branch.
12. Make sure `HPX_VERSION_MAJOR/MINOR/SUBMINOR` in `CMakeLists.txt` contain the correct values. Change them if needed.
13. Remove the examples and benchmarks that will not go into the release from the release branch.
14. This step does not apply to patch releases. Remove features which have been deprecated for at least 2 releases. This involves removing build options which enable those features from the main `CMakeLists.txt` and also deleting all related code and tests from the main source tree.

The general deprecation policy involves a three-step process we have to go through in order to introduce a breaking change

- a. First release cycle: add a build option 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. Switch Buildbot over to test the release branch

- `https://github.com/STELLAR-GROUP/hermione-buildbot/blob/rostan/master/master.cfg`
- branch field in `c['change_source'] = GitPoller`

16. Repeat the following steps until satisfied with the release.

1. Change `HPX_VERSION_TAG` in `CMakeLists.txt` to `-rcN`, where `N` is the current iteration of this step. Start with `-rc1`.
2. 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”.
 - `git tag -a [tag name] -m '[description]'`
 - `git push origin [tag name]`
 - Create a pre-release on GitHub
3. This step is not necessary for patch releases. Notify `hpx-users@stellar.cct.lsu.edu` and `stellar@cct.lsu.edu` of the availability of the release candidate. Ask users to test the candidate by checking out the release candidate tag.
4. Allow at least a week for testing of the release candidate.
 - Use `git merge` when possible, and fall back to `git cherry-pick` when needed. For patch releases `git cherry-pick` is most likely your only choice if there have been significant unrelated changes on master since the previous release.

- Go back to the first step when enough patches have been added.
 - If there are no more patches continue to make the final release.
17. 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.
 18. 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.
 19. Change `HPX_VERSION_TAG` in `CMakeLists.txt` to an empty string.
 20. 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`.
 21. 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 `STELLAR 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]`
 22. Create a release on GitHub
 - Refer to the ‘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.
 23. 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.
 24. 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
```
 25. Update the websites (stellar-group.org²⁵⁸ and stellar.cct.lsu.edu²⁵⁹) 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)
 26. Merge release branch into master.

²⁵⁸ <https://stellar-group.org>

²⁵⁹ <https://stellar.cct.lsu.edu>

27. This step does not apply to patch releases. 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
 1. Create a new “What’s New” section for the docs of the next anticipated release. Set the date to “unreleased”.
 2. Update \$HPX_SOURCE/README.rst
 - Update version (to the about-to-be-released version)
 - Update links to documentation
 - Fix zenodo reference number
 3. Merge new branch containing next version numbers to master, resolve conflicts if necessary.
28. Switch Buildbot back to test the main branch
 - `https://github.com/STELLAR-GROUP/hermione-buildbot/blob/rostan/master/master.cfg`
 - branch field in `c['change_source'] = GitPoller`
29. 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
30. 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](https://www.reddit.com), HPC Wire, Inside HPC, Heise Online, and a CCT press release.
31. 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 CTest²⁶⁰ tool and are executed automatically by buildbot (see [HPX Buildbot Website](#)²⁶¹) on each commit to the HPX Github²⁶² 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](#)²⁶³ tracker with detailed information about the target system.

Running tests manually

Running the tests manually is as easy as typing `make tests && make test`. This will build all tests and run them once the tests are built successfully. After the tests have been built, you can invoke separate tests with the help of the `ctest` command. You can list all available test targets using `make help | grep tests`. Please see the [CTest Documentation](#)²⁶⁴ for further details.

²⁶⁰ <https://gitlab.kitware.com/cmake/community/wikis/doc/ctest/Testing-With-CTest>

²⁶¹ <https://rostan.cct.lsu.edu/>

²⁶² <https://github.com/STELLAR-GROUP/hpx/>

²⁶³ <https://github.com/STELLAR-GROUP/hpx/issues>

²⁶⁴ <https://www.cmake.org/cmake/help/latest/manual/ctest.1.html>

Issue tracker

If you stumble over a bug or missing feature missing feature in *HPX* please submit an issue to our [HPX Issues](https://github.com/STELLAR-GROUP/hpx/issues)²⁶⁵. For more information on how to submit support requests or other means of getting in contact with the developers please see the [Support Website](https://stellar.cct.lsu.edu/support/)²⁶⁶.

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](https://rodamco.cct.lsu.edu/)²⁶⁷. We also run tests on all pull requests using both [CircleCI](https://circleci.com)²⁶⁸ and a combination of [CDash](https://www.kitware.com/cdash/)²⁶⁹ and [pycicle](https://github.com/biddisco/pycicle/)²⁷⁰. You can see the dashboards here: [CircleCI HPX dashboard](https://circleci.com/gh/STELLAR-GROUP/hpx)²⁷¹ and [CDash HPX dashboard](https://cdash.cscs.ch/index.php?project=HPX)²⁷².

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](https://www.docker.com)²⁷³ provides a convenient alternative to quickly get all the dependencies needed to start development of *HPX*. Our testing setup on [CircleCI](https://circleci.com/gh/STELLAR-GROUP/hpx)²⁷⁴ uses a docker image to run all tests.

To get started you need to install [Docker](https://www.docker.com)²⁷⁵ using whatever means is most convenient on your system. Once you have [Docker](https://www.docker.com)²⁷⁶ 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](https://hub.docker.com/r/stellargroup/build_env/)²⁷⁷. 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](https://www.docker.com)²⁷⁸ for more information on using [Docker](https://www.docker.com)²⁷⁹.

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 \(Bind mounts\)](https://docs.docker.com/storage/bind-mounts/)²⁸⁰).

²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues>

²⁶⁶ <https://stellar.cct.lsu.edu/support/>

²⁶⁷ <https://rodamco.cct.lsu.edu/>

²⁶⁸ <https://circleci.com>

²⁶⁹ <https://www.kitware.com/cdash/project/about.html>

²⁷⁰ <https://github.com/biddisco/pycicle/>

²⁷¹ <https://circleci.com/gh/STELLAR-GROUP/hpx>

²⁷² <https://cdash.cscs.ch/index.php?project=HPX>

²⁷³ <https://www.docker.com>

²⁷⁴ <https://circleci.com>

²⁷⁵ <https://www.docker.com>

²⁷⁶ <https://www.docker.com>

²⁷⁷ https://hub.docker.com/r/stellargroup/build_env/

²⁷⁸ <https://docs.docker.com/>

²⁷⁹ <https://www.docker.com>

²⁸⁰ <https://docs.docker.com/storage/bind-mounts/>

2.9.4 Documentation

This documentation is built using [Sphinx](http://www.sphinx-doc.org)²⁸¹, and an automatically generated API reference using [Doxygen](https://www.doxygen.org)²⁸² and [Breathe](https://breathe.readthedocs.io/en/latest)²⁸³.

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](https://www.cmake.org)²⁸⁴ 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.

API documentation

The source code is documented using [Doxygen](https://www.doxygen.org)²⁸⁵. 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`.

²⁸¹ <http://www.sphinx-doc.org>

²⁸² <https://www.doxygen.org>

²⁸³ <https://breathe.readthedocs.io/en/latest>

²⁸⁴ <https://www.cmake.org>

²⁸⁵ <https://www.doxygen.org>

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](https://github.com/STELLAR-GROUP/hpx/issues/3638)²⁸⁶ - Build HPX 1.2 with boost 1.69
- [Issue #3635](https://github.com/STELLAR-GROUP/hpx/issues/3635)²⁸⁷ - Non-deterministic crashing on Stampede2
- [Issue #3550](https://github.com/STELLAR-GROUP/hpx/issues/3550)²⁸⁸ - `1>e:000workhpxsrcthrw_exception.cpp(54): error C2440: '<function-style-cast>': cannot convert from 'boost::system::error_code' to 'hpx::exception'`
- [Issue #3549](https://github.com/STELLAR-GROUP/hpx/issues/3549)²⁸⁹ - HPX 1.2.0 does not build on i686, but release candidate did
- [Issue #3511](https://github.com/STELLAR-GROUP/hpx/issues/3511)²⁹⁰ - Build on s390x fails
- [Issue #3509](https://github.com/STELLAR-GROUP/hpx/issues/3509)²⁹¹ - Build on armv7l fails

Closed pull requests

- [PR #3695](https://github.com/STELLAR-GROUP/hpx/pull/3695)²⁹² - Don't install CMake templates and packaging files

²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3638>

²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3635>

²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3550>

²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3549>

²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3511>

²⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3509>

²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3695>

- PR #3666²⁹³ - Fixing yet another race in future_data
- PR #3663²⁹⁴ - Fixing race between setting and getting the value inside future_data
- PR #3648²⁹⁵ - Adding timestamp option for S390x platform
- PR #3647²⁹⁶ - Blind attempt to fix warnings issued by gcc V9
- PR #3611²⁹⁷ - Include GNUInstallDirs earlier to have it available for subdirectories
- PR #3595²⁹⁸ - Use GNUInstallDirs lib path in pkgconfig config file
- PR #3593²⁹⁹ - Add include(GNUInstallDirs) to HPXMacros.cmake
- PR #3591³⁰⁰ - Fix compilation error on arm7 architecture. Compiles and runs on Fedora 29 on Pi 3.
- PR #3558³⁰¹ - Adding constructor *exception(boost::system::error_code const&)*
- PR #3555³⁰² - cmake: make install locations configurable
- PR #3551³⁰³ - 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`³⁰⁴ or `pkg-config`. The old behaviour can be restored by setting `HPX_WITH_DYNAMIC_HPX_MAIN=OFF` during `CMake`³⁰⁵ configuration. The implementation on Windows is unchanged.
- We have added functionality to allow passing scheduling hints to our schedulers. These will allow us to create executors that for example target a specific NUMA domain or allow for *HPX* threads to be pinned to a particular worker thread.
- We have significantly improved the performance of our futures implementation by making the shared state atomic.
- We have replaced Boostbook by Sphinx for our documentation. This means the documentation is easier to navigate with built-in search and table of contents. We have also added a quick start section and restructured the documentation to be easier to follow for new users.
- We have added a new option to the `--hpx:threads` command line option. It is now possible to use `cores` to tell *HPX* to only use one worker thread per core, unlike the existing option `all` which uses one worker thread

²⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3666>

²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3663>

²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3648>

²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3647>

²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3611>

²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3595>

²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3593>

³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3591>

³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3558>

³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3555>

³⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3551>

³⁰⁴ <https://www.cmake.org>

³⁰⁵ <https://www.cmake.org>

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_HPXM=ON` during *CMake*³⁰⁶ 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*³⁰⁷ 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*³⁰⁸ 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](#)³⁰⁹.
- The following deprecated options have now been completely removed: `HPX_WITH_ASYNC_FUNCTION_COMPATIBILITY`, `HPX_WITH_LOCAL_DATAFLOW`, `HPX_WITH_GENERIC_EXECUTION_POLICY`, `HPX_WITH_BOOST_CHRONO_COMPATIBILITY`, `HPX_WITH_EXECUTOR_COMPATIBILITY`, `HPX_WITH_EXECUTION_POLICY_COMPATIBILITY`, and `HPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY`.

Closed issues

- [Issue #3538](#)³¹⁰ - numa handling incorrect for hwloc 2
- [Issue #3533](#)³¹¹ - Cmake version 3.5.1 does not work (git ff26b35 2018-11-06)

³⁰⁶ <https://www.cmake.org>

³⁰⁷ <https://www.cmake.org>

³⁰⁸ <https://www.cmake.org>

³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3266>

³¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3538>

³¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3533>

- Issue #3526³¹² - Failed building hpx-1.2.0-rc1 on Ubuntu16.04 x86-64 Virtualbox VM
- Issue #3512³¹³ - Build on aarch64 fails
- Issue #3475³¹⁴ - HPX fails to link if the MPI parcellport is enabled
- Issue #3462³¹⁵ - CMake configuration shows a minor and inconsequential failure to create a symlink
- Issue #3461³¹⁶ - Compilation Problems with the most recent Clang
- Issue #3460³¹⁷ - Deadlock when create_partitioner fails (assertion fails) in debug mode
- Issue #3455³¹⁸ - HPX build failing with HWLOC errors on POWER8 with hwloc 1.8
- Issue #3438³¹⁹ - HPX no longer builds on IBM POWER8
- Issue #3426³²⁰ - hpx build failed on MacOS
- Issue #3424³²¹ - CircleCI builds broken for forked repositories
- Issue #3422³²² - Benchmarks in tests.performance.local are not run nightly
- Issue #3408³²³ - CMake Targets for HPX
- Issue #3399³²⁴ - processing unit out of bounds
- Issue #3395³²⁵ - Floating point bug in hpx/runtime/threads/policies/scheduler_base.hpp
- Issue #3378³²⁶ - compile error with lcos::communicator
- Issue #3376³²⁷ - Failed to build HPX with APEX using clang
- Issue #3366³²⁸ - Adapted Safe_Object example fails for -hpx:threads > 1
- Issue #3360³²⁹ - Segmentation fault when passing component id as parameter
- Issue #3358³³⁰ - HPX runtime hangs after multiple (~thousands) start-stop sequences
- Issue #3352³³¹ - Support TCP provider in libfabric ParcelPort
- Issue #3342³³² - undefined reference to __atomic_load_16
- Issue #3339³³³ - setting command line options/flags from init cfg is not obvious
- Issue #3325³³⁴ - AGAS migrates components prematurely

³¹² <https://github.com/STELLAR-GROUP/hpx/issues/3526>

³¹³ <https://github.com/STELLAR-GROUP/hpx/issues/3512>

³¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3475>

³¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3462>

³¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3461>

³¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3460>

³¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3455>

³¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3438>

³²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3426>

³²¹ <https://github.com/STELLAR-GROUP/hpx/issues/3424>

³²² <https://github.com/STELLAR-GROUP/hpx/issues/3422>

³²³ <https://github.com/STELLAR-GROUP/hpx/issues/3408>

³²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3399>

³²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3395>

³²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3378>

³²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3376>

³²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3366>

³²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3360>

³³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3358>

³³¹ <https://github.com/STELLAR-GROUP/hpx/issues/3352>

³³² <https://github.com/STELLAR-GROUP/hpx/issues/3342>

³³³ <https://github.com/STELLAR-GROUP/hpx/issues/3339>

³³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3325>

- Issue #3321³³⁵ - hpx bad_parameter handling is awful
- Issue #3318³³⁶ - Benchmarks fail to build with C++11
- Issue #3304³³⁷ - hpx::threads::run_as_hpx_thread does not properly handle exceptions
- Issue #3300³³⁸ - Setting pu step or offset results in no threads in default pool
- Issue #3297³³⁹ - Crash with APEX when running Phylaxx lra_csv with > 1 thread
- Issue #3296³⁴⁰ - Building HPX with APEX configuration gives compiler warnings
- Issue #3290³⁴¹ - make tests failing at hello_world_component
- Issue #3285³⁴² - possible compilation error when “using namespace std;” is defined before including “hpx” headers files
- Issue #3280³⁴³ - HPX fails on OSX
- Issue #3272³⁴⁴ - CircleCI does not upload generated docker image any more
- Issue #3270³⁴⁵ - Error when compiling CUDA examples
- Issue #3267³⁴⁶ - tests.unit.host_.block_allocator fails occasionally
- Issue #3264³⁴⁷ - Possible move to Sphinx for documentation
- Issue #3263³⁴⁸ - Documentation improvements
- Issue #3259³⁴⁹ - set_parcel_write_handler test fails occasionally
- Issue #3258³⁵⁰ - Links to source code in documentation are broken
- Issue #3247³⁵¹ - Rare tests.unit.host_.block_allocator test failure on 1.1.0-rc1
- Issue #3244³⁵² - Slowing down and speeding up an interval_timer
- Issue #3215³⁵³ - Cannot build both tests and examples on MSVC with pseudo-dependencies enabled
- Issue #3195³⁵⁴ - Unnecessary customization point route causing performance penalty
- Issue #3088³⁵⁵ - A strange thing in parallel::sort.
- Issue #2650³⁵⁶ - libfabric support for passive endpoints
- Issue #1205³⁵⁷ - TSS is broken

³³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3321>

³³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3318>

³³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3304>

³³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3300>

³³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3297>

³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3296>

³⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/3290>

³⁴² <https://github.com/STELLAR-GROUP/hpx/issues/3285>

³⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/3280>

³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3272>

³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3270>

³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3267>

³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3264>

³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3263>

³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3259>

³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3258>

³⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/3247>

³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/3244>

³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/3215>

³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3195>

³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3088>

³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2650>

³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1205>

Closed pull requests

- PR #3542³⁵⁸ - Fix numa lookup from pu when using hwloc 2.x
- PR #3541³⁵⁹ - Fixing the build system of the MPI parcelport
- PR #3540³⁶⁰ - Updating HPX people section
- PR #3539³⁶¹ - Splitting test to avoid OOM on CircleCI
- PR #3537³⁶² - Fix guided exec
- PR #3536³⁶³ - Updating grants which support the LSU team
- PR #3535³⁶⁴ - Fix hiding of docker credentials
- PR #3534³⁶⁵ - Fixing #3533
- PR #3532³⁶⁶ - fixing minor doc typo `-hpx:print-counter-at` arg
- PR #3530³⁶⁷ - Changing APEX default tag to v2.1.0
- PR #3529³⁶⁸ - Remove leftover security options and documentation
- PR #3528³⁶⁹ - Fix hwloc version check
- PR #3524³⁷⁰ - Do not build guided pool examples with older GCC compilers
- PR #3523³⁷¹ - Fix logging regression
- PR #3522³⁷² - Fix more warnings
- PR #3521³⁷³ - Fixing argument handling in induction and reduction clauses for `parallel::for_loop`
- PR #3520³⁷⁴ - Remove docs symlink and versioned docs folders
- PR #3519³⁷⁵ - hpxMP release
- PR #3518³⁷⁶ - Change all steps to use new docker image on CircleCI
- PR #3516³⁷⁷ - Drop usage of deprecated facilities removed in C++17
- PR #3515³⁷⁸ - Remove remaining uses of Boost.TypeTraits
- PR #3513³⁷⁹ - Fixing a CMake problem when trying to use libfabric

³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3542>

³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3541>

³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3540>

³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3539>

³⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3537>

³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3536>

³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3535>

³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3534>

³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3532>

³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3530>

³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3529>

³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3528>

³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3524>

³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3523>

³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3522>

³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3521>

³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3520>

³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3519>

³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3518>

³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3516>

³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3515>

³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3513>

- [PR #3508](#)³⁸⁰ - Remove memory_block component
- [PR #3507](#)³⁸¹ - Propagating the MPI compile definitions to all relevant targets
- [PR #3503](#)³⁸² - Update documentation colors and logo
- [PR #3502](#)³⁸³ - Fix bogus ‘throws’ bindings in scheduled_thread_pool_impl
- [PR #3501](#)³⁸⁴ - Split parallel::remove_if tests to avoid OOM on CircleCI
- [PR #3500](#)³⁸⁵ - Support NONAMEPREFIX in add_hpx_library()
- [PR #3497](#)³⁸⁶ - Note that cuda support requires cmake 3.9
- [PR #3495](#)³⁸⁷ - Fixing dataflow
- [PR #3493](#)³⁸⁸ - Remove deprecated options for 1.2.0 part 2
- [PR #3492](#)³⁸⁹ - Add CUDA_LINK_LIBRARIES_KEYWORD to allow PRIVATE keyword in linkage t...
- [PR #3491](#)³⁹⁰ - Changing Base docker image
- [PR #3490](#)³⁹¹ - Don't create tasks immediately with hpx::apply
- [PR #3489](#)³⁹² - Remove deprecated options for 1.2.0
- [PR #3488](#)³⁹³ - Revert “Use BUILD_INTERFACE generator expression to fix cmake flag exports”
- [PR #3487](#)³⁹⁴ - Revert “Fixing type attribute warning for transfer_action”
- [PR #3485](#)³⁹⁵ - Use BUILD_INTERFACE generator expression to fix cmake flag exports
- [PR #3483](#)³⁹⁶ - Fixing type attribute warning for transfer_action
- [PR #3481](#)³⁹⁷ - Remove unused variables
- [PR #3480](#)³⁹⁸ - Towards a more lightweight transfer action
- [PR #3479](#)³⁹⁹ - Fix FLAGS - Use correct version of target_compile_options
- [PR #3478](#)⁴⁰⁰ - Making sure the application's exit code is properly propagated back to the OS
- [PR #3476](#)⁴⁰¹ - Don't print docker credentials as part of the environment.
- [PR #3473](#)⁴⁰² - Fixing invalid cmake code if no jemalloc prefix was given

³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3508>

³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3507>

³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3503>

³⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3502>

³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3501>

³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3500>

³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3497>

³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3495>

³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3493>

³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3492>

³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3491>

³⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3490>

³⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3489>

³⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3488>

³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3487>

³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3485>

³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3483>

³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3481>

³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3480>

³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3479>

⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3478>

⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3476>

⁴⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3473>

- PR #3472⁴⁰³ - Attempting to work around recent clang test compilation failures
- PR #3471⁴⁰⁴ - Enable jemalloc on windows
- PR #3470⁴⁰⁵ - Updates readme
- PR #3468⁴⁰⁶ - Avoid hang if there is an exception thrown during startup
- PR #3467⁴⁰⁷ - Add compiler specific fallthrough attributes if C++17 attribute is not available
- PR #3466⁴⁰⁸ - - bugfix : fix compilation with llvm-7.0
- PR #3465⁴⁰⁹ - This patch adds various optimizations extracted from the thread_local_allocator work
- PR #3464⁴¹⁰ - Check for forked repos in CircleCI docker push step
- PR #3463⁴¹¹ - - cmake : create the parent directory before symlinking
- PR #3459⁴¹² - Remove unused/incomplete functionality from util/logging
- PR #3458⁴¹³ - Fix a problem with scope of CMAKE_CXX_FLAGS and hpx_add_compile_flag
- PR #3457⁴¹⁴ - Fixing more size_t -> int16_t (and similar) warnings
- PR #3456⁴¹⁵ - Add #ifdefs to topology.cpp to support old hwloc versions again
- PR #3454⁴¹⁶ - Fixing warnings related to silent conversion of size_t -> int16_t
- PR #3451⁴¹⁷ - Add examples as unit tests
- PR #3450⁴¹⁸ - Constexpr-fying bind and other functional facilities
- PR #3446⁴¹⁹ - Fix some thread suspension timeouts
- PR #3445⁴²⁰ - Fix various warnings
- PR #3443⁴²¹ - Only enable service pool config options if pools are enabled
- PR #3441⁴²² - Fix missing closing brackets in documentation
- PR #3439⁴²³ - Use correct MPI CXX libraries for MPI parcelport
- PR #3436⁴²⁴ - Add projection function to find_* (and fix very bad bug)
- PR #3435⁴²⁵ - Fixing 1205

⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3472>

⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3471>

⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3470>

⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3468>

⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3467>

⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3466>

⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3465>

⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3464>

⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3463>

⁴¹² <https://github.com/STELLAR-GROUP/hpx/pull/3459>

⁴¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3458>

⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3457>

⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3456>

⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3454>

⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3451>

⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3450>

⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3446>

⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3445>

⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3443>

⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/3441>

⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/3439>

⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3436>

⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3435>

- PR #3434⁴²⁶ - Fix threads cores
- PR #3433⁴²⁷ - Add Heise Online to release announcement list
- PR #3432⁴²⁸ - Don't track task dependencies for distributed runs
- PR #3431⁴²⁹ - Circle CI setting changes for hpxMP
- PR #3430⁴³⁰ - Fix unused params warning
- PR #3429⁴³¹ - One thread per core
- PR #3428⁴³² - This suppresses a deprecation warning that is being issued by MSVC 19.15.26726
- PR #3427⁴³³ - Fixes #3426
- PR #3425⁴³⁴ - Use source cache and workspace between job steps on CircleCI
- PR #3421⁴³⁵ - Add CDash timing output to future overhead test (for graphs)
- PR #3420⁴³⁶ - Add guided_pool_executor
- PR #3419⁴³⁷ - Fix typo in CircleCI config
- PR #3418⁴³⁸ - Add sphinx documentation
- PR #3415⁴³⁹ - Scheduler NUMA hint and shared priority scheduler
- PR #3414⁴⁴⁰ - Adding step to synchronize the APEX release
- PR #3413⁴⁴¹ - Fixing multiple defines of APEX_HAVE_HPX
- PR #3412⁴⁴² - Fixes linking with libhpx_wrap error with BSD and Windows based systems
- PR #3410⁴⁴³ - Fix typo in CMakeLists.txt
- PR #3409⁴⁴⁴ - Fix brackets and indentation in existing_performance_counters.qbk
- PR #3407⁴⁴⁵ - Fix unused param and extra ; warnings emitted by gcc 8.x
- PR #3406⁴⁴⁶ - Adding thread local allocator and use it for future shared states
- PR #3405⁴⁴⁷ - Adding DHPX_HAVE_THREAD_LOCAL_STORAGE=ON to builds
- PR #3404⁴⁴⁸ - fixing multiple definition of main() in linux

⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3434>

⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3433>

⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3432>

⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3431>

⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3430>

⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3429>

⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/3428>

⁴³³ <https://github.com/STELLAR-GROUP/hpx/pull/3427>

⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3425>

⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3421>

⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3420>

⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3419>

⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3418>

⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3415>

⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3414>

⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3413>

⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3412>

⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3410>

⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3409>

⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3407>

⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3406>

⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3405>

⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3404>

- PR #3402⁴⁴⁹ - Allow debug option to be enabled only for Linux systems with dynamic main on
- PR #3401⁴⁵⁰ - Fix cuda_future_helper.h when compiling with C++11
- PR #3400⁴⁵¹ - Fix floating point exception scheduler_base idle backoff
- PR #3398⁴⁵² - Atomic future state
- PR #3397⁴⁵³ - Fixing code for older gcc versions
- PR #3396⁴⁵⁴ - Allowing to register thread event functions (start/stop/error)
- PR #3394⁴⁵⁵ - Fix small mistake in primary_namespace_server.cpp
- PR #3393⁴⁵⁶ - Explicitly instantiate configured schedulers
- PR #3392⁴⁵⁷ - Add performance counters background overhead and background work duration
- PR #3391⁴⁵⁸ - Adapt integration of HPXMP to latest build system changes
- PR #3390⁴⁵⁹ - Make AGAS measurements optional
- PR #3389⁴⁶⁰ - Fix deadlock during shutdown
- PR #3388⁴⁶¹ - Add several functionalities allowing to optimize synchronous action invocation
- PR #3387⁴⁶² - Add cmake option to opt out of fail-compile tests
- PR #3386⁴⁶³ - Adding support for boost::container::small_vector to dataflow
- PR #3385⁴⁶⁴ - Adds Debug option for hpx initializing from main
- PR #3384⁴⁶⁵ - This hopefully fixes two tests that occasionally fail
- PR #3383⁴⁶⁶ - Making sure thread local storage is enable for hpxMP
- PR #3382⁴⁶⁷ - Fix usage of HPX_CAPTURE together with default value capture [=]
- PR #3381⁴⁶⁸ - Replace undefined instantiations of uniform_int_distribution
- PR #3380⁴⁶⁹ - Add missing semicolons to uses of HPX_COMPILER_FENCE
- PR #3379⁴⁷⁰ - Fixing #3378
- PR #3377⁴⁷¹ - Adding build system support to integrate hpxmp into hpx at the user's machine

⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3402>

⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3401>

⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3400>

⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3398>

⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3397>

⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3396>

⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3394>

⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3393>

⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3392>

⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3391>

⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3390>

⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3389>

⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3388>

⁴⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3387>

⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3386>

⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3385>

⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3384>

⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3383>

⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3382>

⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3381>

⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3380>

⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3379>

⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3377>

- PR #3375⁴⁷² - Replacing wrapper for `__libc_start_main` with `main`
- PR #3374⁴⁷³ - Adds `hpx_wrap` to `HPX_LINK_LIBRARIES` which links only when specified.
- PR #3373⁴⁷⁴ - Forcing cache settings in `HPXConfig.cmake` to guarantee updated values
- PR #3372⁴⁷⁵ - Fix some more c++11 build problems
- PR #3371⁴⁷⁶ - Adds `HPX_LINKER_FLAGS` to HPX applications without editing their source codes
- PR #3370⁴⁷⁷ - `util::format`: add `type_specifier<>` specializations for `%!s(MISSING)` and `%!l(MISSING)s`
- PR #3369⁴⁷⁸ - Adding configuration option to allow explicit disable of the new `hpx_main` feature on Linux
- PR #3368⁴⁷⁹ - Updates doc with recent `hpx_wrap` implementation
- PR #3367⁴⁸⁰ - Adds Mac OS implementation to `hpx_main.hpp`
- PR #3365⁴⁸¹ - Fix order of `hpx` libs in `HPX_CONF_LIBRARIES`.
- PR #3363⁴⁸² - Apex fixing null wrapper
- PR #3361⁴⁸³ - Making sure all parcels get destroyed on an HPX thread (TCP pp)
- PR #3359⁴⁸⁴ - Feature/improve error for compiler
- PR #3357⁴⁸⁵ - Static/dynamic executable implementation
- PR #3355⁴⁸⁶ - Reverting changes introduced by #3283 as those make applications hang
- PR #3354⁴⁸⁷ - Add external dependencies to `HPX_LIBRARY_DIR`
- PR #3353⁴⁸⁸ - Fix `libfabric tcp`
- PR #3351⁴⁸⁹ - Move obsolete header to tests directory.
- PR #3350⁴⁹⁰ - Renaming two functions to avoid problem described in #3285
- PR #3349⁴⁹¹ - Make idle backoff exponential with maximum sleep time
- PR #3347⁴⁹² - Replace `simple_component*` with `component*` in the Documentation
- PR #3346⁴⁹³ - Fix `CMakeLists.txt` example in quick start
- PR #3345⁴⁹⁴ - Fix automatic setting of `HPX_MORE_THAN_64_THREADS`

⁴⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3375>

⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3374>

⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3373>

⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3372>

⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3371>

⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3370>

⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3369>

⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3368>

⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3367>

⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3365>

⁴⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3363>

⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3361>

⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3359>

⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3357>

⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3355>

⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3354>

⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3353>

⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3351>

⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3350>

⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3349>

⁴⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3347>

⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3346>

⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3345>

- PR #3344⁴⁹⁵ - Reduce amount of information printed for unknown command line options
- PR #3343⁴⁹⁶ - Safeguard HPX against destruction in global contexts
- PR #3341⁴⁹⁷ - Allowing for all command line options to be used as configuration settings
- PR #3340⁴⁹⁸ - Always convert inspect results to JUnit XML
- PR #3336⁴⁹⁹ - Only run docker push on master on CircleCI
- PR #3335⁵⁰⁰ - Update description of hpx.os_threads config parameter.
- PR #3334⁵⁰¹ - Making sure early logging settings don't get mixed with others
- PR #3333⁵⁰² - Update CMake links and versions in documentation
- PR #3332⁵⁰³ - Add notes on target suffixes to CMake documentation
- PR #3331⁵⁰⁴ - Add quickstart section to documentation
- PR #3330⁵⁰⁵ - Rename resource_partitioner test to avoid conflicts with pseudodependencies
- PR #3328⁵⁰⁶ - Making sure object is pinned while executing actions, even if action returns a future
- PR #3327⁵⁰⁷ - Add missing std::forward to tuple.hpp
- PR #3326⁵⁰⁸ - Make sure logging is up and running while modules are being discovered.
- PR #3324⁵⁰⁹ - Replace C++14 overload of std::equal with C++11 code.
- PR #3323⁵¹⁰ - Fix a missing apex thread data (wrapper) initialization
- PR #3320⁵¹¹ - Adding support for -std=c++2a (define *HPX_WITH_CXX2A=On*)
- PR #3319⁵¹² - Replacing C++14 feature with equivalent C++11 code
- PR #3317⁵¹³ - Fix compilation with VS 15.7.1 and /std:c++latest
- PR #3316⁵¹⁴ - Fix includes for 1d_stencil_*_omp examples
- PR #3314⁵¹⁵ - Remove some unused parameter warnings
- PR #3313⁵¹⁶ - Fix pu-step and pu-offset command line options
- PR #3312⁵¹⁷ - Add conversion of inspect reports to JUnit XML

⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3344>

⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3343>

⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3341>

⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3340>

⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3336>

⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3335>

⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3334>

⁵⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3333>

⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3332>

⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3331>

⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3330>

⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3328>

⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3327>

⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3326>

⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3324>

⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3323>

⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3320>

⁵¹² <https://github.com/STELLAR-GROUP/hpx/pull/3319>

⁵¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3317>

⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3316>

⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3314>

⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3313>

⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3312>

- PR #3311⁵¹⁸ - Fix escaping of closing braces in format specification syntax
- PR #3310⁵¹⁹ - Don't overwrite user settings with defaults in registration database
- PR #3309⁵²⁰ - Fixing potential stack overflow for dataflow
- PR #3308⁵²¹ - This updates the .clang-format configuration file to utilize newer features
- PR #3306⁵²² - Marking migratable objects in their gid to allow not handling migration in AGAS
- PR #3305⁵²³ - Add proper exception handling to run_as_hpx_thread
- PR #3303⁵²⁴ - Changed std::rand to a better inbuilt PRNG Generator
- PR #3302⁵²⁵ - All non-migratable (simple) components now encode their lva and component type in their gid
- PR #3301⁵²⁶ - Add nullptr_t overloads to resource partitioner
- PR #3298⁵²⁷ - Apex task wrapper memory bug
- PR #3295⁵²⁸ - Fix mistakes after merge of CircleCI config
- PR #3294⁵²⁹ - Fix partitioned vector include in partitioned_vector_find tests
- PR #3293⁵³⁰ - Adding emplace support to promise and make_ready_future
- PR #3292⁵³¹ - Add new cuda kernel synchronization with hpx::future demo
- PR #3291⁵³² - Fixes #3290
- PR #3289⁵³³ - Fixing Docker image creation
- PR #3288⁵³⁴ - Avoid allocating shared state for wait_all
- PR #3287⁵³⁵ - Fixing /scheduler/utilization/instantaneous performance counter
- PR #3286⁵³⁶ - dataflow() and future::then() use sync policy where possible
- PR #3284⁵³⁷ - Background thread can use relaxed atomics to manipulate thread state
- PR #3283⁵³⁸ - Do not unwrap ready future
- PR #3282⁵³⁹ - Fix virtual method override warnings in static schedulers
- PR #3281⁵⁴⁰ - Disable set_area_membind_nodeuset for OSX

⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3311>

⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3310>

⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3309>

⁵²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3308>

⁵²² <https://github.com/STELLAR-GROUP/hpx/pull/3306>

⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/3305>

⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3303>

⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3302>

⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3301>

⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3298>

⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3295>

⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3294>

⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3293>

⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3292>

⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/3291>

⁵³³ <https://github.com/STELLAR-GROUP/hpx/pull/3289>

⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3288>

⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3287>

⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3286>

⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3284>

⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3283>

⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3282>

⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3281>

- PR #3279⁵⁴¹ - Add two variations to the future_overhead benchmark
- PR #3278⁵⁴² - Fix circleci workspace
- PR #3277⁵⁴³ - Support external plugins
- PR #3276⁵⁴⁴ - Fix missing parenthesis in hello_compute.cu.
- PR #3274⁵⁴⁵ - Reinit counters synchronously in reinit_counters test
- PR #3273⁵⁴⁶ - Splitting tests to avoid compiler OOM
- PR #3271⁵⁴⁷ - Remove leftover code from context_generic_context.hpp
- PR #3269⁵⁴⁸ - Fix bulk_construct with count = 0
- PR #3268⁵⁴⁹ - Replace constexpr with HPX_CXX14_CONSTEXPR and HPX_CONSTEXPR
- PR #3266⁵⁵⁰ - Replace boost::format with custom sprintf-based implementation
- PR #3265⁵⁵¹ - Split parallel tests on CircleCI
- PR #3262⁵⁵² - Making sure documentation correctly links to source files
- PR #3261⁵⁵³ - Apex refactoring fix rebind
- PR #3260⁵⁵⁴ - Isolate performance counter parser into a separate TU
- PR #3256⁵⁵⁵ - Post 1.1.0 version bumps
- PR #3254⁵⁵⁶ - Adding trait for actions allowing to make runtime decision on whether to execute it directly
- PR #3253⁵⁵⁷ - Bump minimal supported Boost to 1.58.0
- PR #3251⁵⁵⁸ - Adds new feature: changing interval used in interval_timer (issue 3244)
- PR #3239⁵⁵⁹ - Changing std::rand() to a better inbuilt PRNG generator.
- PR #3234⁵⁶⁰ - Disable background thread when networking is off
- PR #3232⁵⁶¹ - Clean up suspension tests
- PR #3230⁵⁶² - Add optional scheduler mode parameter to create_thread_pool function
- PR #3228⁵⁶³ - Allow suspension also on static schedulers

⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3279>

⁵⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3278>

⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3277>

⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3276>

⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3274>

⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3273>

⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3271>

⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3269>

⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3268>

⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3266>

⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3265>

⁵⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3262>

⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3261>

⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3260>

⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3256>

⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3254>

⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3253>

⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3251>

⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3239>

⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3234>

⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3232>

⁵⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3230>

⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3228>

- [PR #3163](#)⁵⁶⁴ - libfabric parcelport w/o HPX_PARCELPORT_LIBFABRIC_ENDPOINT_RDM
- [PR #3036](#)⁵⁶⁵ - 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 now be suspended/resumed independently. This functionality helps in running *HPX* concurrently to code that is directly relying on [OpenMP](#)⁵⁶⁶ and/or [MPI](#)⁵⁶⁷.
- 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](#)⁵⁶⁸. 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](#)⁵⁶⁹ 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](#)⁵⁷⁰.
- The parameter sequence for the `hpx::parallel::inclusive_scan` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17. The old overloads can be still enabled at configure time by passing `-DHPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY=On` to CMake.
- Added a helper facility `hpx::local_new` which is equivalent to `hpx::new_` except that it creates components locally only. As a consequence, the used component constructor may accept non-serializable argument types and/or non-const references or pointers.
- Removed the (broken) component type `hpx::lcos::queue<T>`. The old type is still available at configure time by passing `-DHPX_WITH_QUEUE_COMPATIBILITY=On` to CMake.
- The parallel algorithms adopted for C++17 restrict the iterator categories usable with those to at least forward iterators. Our implementation of the parallel algorithms was supporting input iterators (and output iterators) as well by simply falling back to sequential execution. We have now made our implementations conforming by requiring at least forward iterators. In order to enable the old behavior use the the compatibility option `-DHPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT=On` on the [CMake](#)⁵⁷¹ command line.

⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3163>

⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3036>

⁵⁶⁶ <https://openmp.org/wp/>

⁵⁶⁷ https://en.wikipedia.org/wiki/Message_Passing_Interface

⁵⁶⁸ <http://www.open-std.org/jtc1/sc22/wg21>

⁵⁶⁹ <https://www.cmake.org>

⁵⁷⁰ <https://www.cmake.org>

⁵⁷¹ <https://www.cmake.org>

- 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 CMake⁵⁷² 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 #3250⁵⁷³ - Apex refactoring with guides
- PR #3249⁵⁷⁴ - Updating People.qbk
- PR #3246⁵⁷⁵ - Assorted fixes for CUDA
- PR #3245⁵⁷⁶ - Apex refactoring with guides
- PR #3242⁵⁷⁷ - Modify task counting in `thread_queue.hpp`

⁵⁷² <https://www.cmake.org>

⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3250>

⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3249>

⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3246>

⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3245>

⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3242>

- PR #3240⁵⁷⁸ - Fixed typos
- PR #3238⁵⁷⁹ - Readding accidentally removed `std::abort`
- PR #3237⁵⁸⁰ - Adding Pipeline example
- PR #3236⁵⁸¹ - Fixing `memory_block`
- PR #3233⁵⁸² - Make `schedule_thread` take suspended threads into account
- Issue #3226⁵⁸³ - `memory_block` is breaking, signaling SIGSEGV on a thread on creation and freeing
- PR #3225⁵⁸⁴ - Applying quick fix for `hwloc-2.0`
- Issue #3224⁵⁸⁵ - HPX counters crashing the application
- PR #3223⁵⁸⁶ - Fix returns when setting config entries
- Issue #3222⁵⁸⁷ - Errors linking `libhpx.so`
- Issue #3221⁵⁸⁸ - HPX on Mac OS X with `HWLoc 2.0.0` fails to run
- PR #3216⁵⁸⁹ - Reorder a variadic array to satisfy VS 2017 15.6
- PR #3214⁵⁹⁰ - Changed `prerequisites.qbk` to avoid confusion while building boost
- PR #3213⁵⁹¹ - Relax locks for thread suspension to avoid holding locks when yielding
- PR #3212⁵⁹² - Fix check in `sequenced_executor` test
- PR #3211⁵⁹³ - Use `preinit_array` to set `argc/argv` in `init_globally` example
- PR #3210⁵⁹⁴ - Adapted `parallel::{search | search_n}` for Ranges TS (see #1668)
- PR #3209⁵⁹⁵ - Fix locking problems during shutdown
- Issue #3208⁵⁹⁶ - `init_globally` throwing a run-time error
- PR #3206⁵⁹⁷ - Addition of new arithmetic performance counter “Count”
- PR #3205⁵⁹⁸ - Fixing return type calculation for `bulk_then_execute`
- PR #3204⁵⁹⁹ - Changing `std::rand()` to a better inbuilt PRNG generator
- PR #3203⁶⁰⁰ - Resolving problems during shutdown for VS2015

⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3240>

⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3238>

⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3237>

⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3236>

⁵⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3233>

⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/3226>

⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3225>

⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3224>

⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3223>

⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3222>

⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3221>

⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3216>

⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3214>

⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3213>

⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3212>

⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3211>

⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3210>

⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3209>

⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3208>

⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3206>

⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3205>

⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3204>

⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3203>

- PR #3202⁶⁰¹ - Making sure resource partitioner is not accessed if its not valid
- PR #3201⁶⁰² - Fixing optional::swap
- Issue #3200⁶⁰³ - hpx::util::optional fails
- PR #3199⁶⁰⁴ - Fix sliding_semaphore test
- PR #3198⁶⁰⁵ - Set pre_main status before launching run_helper
- PR #3197⁶⁰⁶ - Update README.rst
- PR #3194⁶⁰⁷ - parallel::{fillfill_n} updated for Ranges TS
- PR #3193⁶⁰⁸ - Updating Runtime.cpp by adding correct description of Performance counters during register
- PR #3191⁶⁰⁹ - Fix sliding_semaphore_2338 test
- PR #3190⁶¹⁰ - Topology improvements
- PR #3189⁶¹¹ - Deleting one include of median from BOOST library to arithmetics_counter file
- PR #3188⁶¹² - Optionally disable printing of diagnostics during terminate
- PR #3187⁶¹³ - Suppressing cmake warning issued by cmake > V3.11
- PR #3185⁶¹⁴ - Remove unused scoped_unlock, unlock_guard_try
- PR #3184⁶¹⁵ - Fix nqueen example
- PR #3183⁶¹⁶ - Add runtime start/stop, resume/suspend and OpenMP benchmarks
- Issue #3182⁶¹⁷ - bulk_then_execute has unexpected return type/does not compile
- Issue #3181⁶¹⁸ - hwloc 2.0 breaks topo class and cannot be used
- Issue #3180⁶¹⁹ - Schedulers that don't support suspend/resume are unusable
- PR #3179⁶²⁰ - Various minor changes to support FLeCSI
- PR #3178⁶²¹ - Fix #3124
- PR #3177⁶²² - Removed allgather
- PR #3176⁶²³ - Fixed Documentation for "using_hpx_pkgconfig"

⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3202>

⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3201>

⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/3200>

⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3199>

⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3198>

⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3197>

⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3194>

⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3193>

⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3191>

⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3190>

⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3189>

⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/3188>

⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3187>

⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3185>

⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3184>

⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3183>

⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3182>

⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3181>

⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3180>

⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3179>

⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3178>

⁶²² <https://github.com/STELLAR-GROUP/hpx/pull/3177>

⁶²³ <https://github.com/STELLAR-GROUP/hpx/pull/3176>

- PR #3174⁶²⁴ - Add `hpx::iostreams::ostream` overload to `format_to`
- PR #3172⁶²⁵ - Fix lifo queue backend
- PR #3171⁶²⁶ - adding the missing `unset()` function to `cpu_mask()` for case of more than 64 threads
- PR #3170⁶²⁷ - Add cmake flag `-DHPX_WITH_FAULT_TOLERANCE=ON` (OFF by default)
- PR #3169⁶²⁸ - Adapted `parallel::{count|count_if}` for Ranges TS (see #1668)
- PR #3168⁶²⁹ - Changing used namespace for seq execution policy
- Issue #3167⁶³⁰ - Update GSoC projects
- Issue #3166⁶³¹ - Application (Octotiger) gets stuck on `hpx::finalize` when only using one thread
- Issue #3165⁶³² - Compilation of parallel algorithms with `HPX_WITH_DATAPAR` is broken
- PR #3164⁶³³ - Fixing component migration
- PR #3162⁶³⁴ - `regex_from_pattern`: escape regex special characters to avoid misinterpretation
- Issue #3161⁶³⁵ - Building HPX with `hwloc 2.0.0` fails
- PR #3160⁶³⁶ - Fixing the handling of quoted command line arguments.
- PR #3158⁶³⁷ - Fixing a race with timed suspension (second attempt)
- PR #3157⁶³⁸ - Revert “Fixing a race with timed suspension”
- PR #3156⁶³⁹ - Fixing serialization of classes with incompatible `serialize` signature
- PR #3154⁶⁴⁰ - More refactorings based on clang-tidy reports
- PR #3153⁶⁴¹ - Fixing a race with timed suspension
- PR #3152⁶⁴² - Documentation for runtime suspension
- PR #3151⁶⁴³ - Use `small_vector` only from boost version 1.59 onwards
- PR #3150⁶⁴⁴ - Avoiding more stack overflows
- PR #3148⁶⁴⁵ - Refactoring `component_base` and `base_action/transfer_base_action`
- PR #3147⁶⁴⁶ - Move `yield_while` out of `detail` namespace and into own file

⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3174>

⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3172>

⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3171>

⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3170>

⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3169>

⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3168>

⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3167>

⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/3166>

⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/3165>

⁶³³ <https://github.com/STELLAR-GROUP/hpx/pull/3164>

⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3162>

⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3161>

⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3160>

⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3158>

⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3157>

⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3156>

⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3154>

⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3153>

⁶⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3152>

⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3151>

⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3150>

⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3148>

⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3147>

- PR #3145⁶⁴⁷ - Remove a leftover of the cxx11 std array cleanup
- PR #3144⁶⁴⁸ - Minor changes to how actions are executed
- PR #3143⁶⁴⁹ - Fix stack overhead
- PR #3142⁶⁵⁰ - Fix typo in config.hpp
- PR #3141⁶⁵¹ - Fixing small_vector compatibility with older boost version
- PR #3140⁶⁵² - is_heap_text fix
- Issue #3139⁶⁵³ - Error in is_heap_tests.hpp
- PR #3138⁶⁵⁴ - Partially reverting #3126
- PR #3137⁶⁵⁵ - Suspend speedup
- PR #3136⁶⁵⁶ - Revert “Fixing #2325”
- PR #3135⁶⁵⁷ - Improving destruction of threads
- Issue #3134⁶⁵⁸ - HPX_SERIALIZATION_SPLIT_FREE does not stop compiler from looking for serialize() method
- PR #3133⁶⁵⁹ - Make hwloc compulsory
- PR #3132⁶⁶⁰ - Update CXX14 constexpr feature test
- PR #3131⁶⁶¹ - Fixing #2325
- PR #3130⁶⁶² - Avoid completion handler allocation
- PR #3129⁶⁶³ - Suspend runtime
- PR #3128⁶⁶⁴ - Make docbook dtd and xsl path names consistent
- PR #3127⁶⁶⁵ - Add hpx::start nullptr overloads
- PR #3126⁶⁶⁶ - Cleaning up coroutine implementation
- PR #3125⁶⁶⁷ - Replacing nullptr with hpx::threads::invalid_thread_id
- Issue #3124⁶⁶⁸ - Add hello_world_component to CI builds
- PR #3123⁶⁶⁹ - Add new constructor.

⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3145>

⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3144>

⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3143>

⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3142>

⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3141>

⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3140>

⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/3139>

⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3138>

⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3137>

⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3136>

⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3135>

⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3134>

⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3133>

⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3132>

⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3131>

⁶⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3130>

⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3129>

⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3128>

⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3127>

⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3126>

⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3125>

⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3124>

⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3123>

- PR #3122⁶⁷⁰ - Fixing #3121
- Issue #3121⁶⁷¹ - HPX_SMT_PAUSE is broken on non-x86 platforms when `__GNUC__` is defined
- PR #3120⁶⁷² - Don't use `boost::intrusive_ptr` for `thread_id_type`
- PR #3119⁶⁷³ - Disable default executor compatibility with V1 executors
- PR #3118⁶⁷⁴ - Adding `performance_counter::reinit` to allow for dynamically changing counter sets
- PR #3117⁶⁷⁵ - Replace uses of `boost/experimental::optional` with `util::optional`
- PR #3116⁶⁷⁶ - Moving background thread APEX timer #2980
- PR #3115⁶⁷⁷ - Fixing race condition in channel test
- PR #3114⁶⁷⁸ - Avoid using `util::function` for thread function wrappers
- PR #3113⁶⁷⁹ - cmake V3.10.2 has changed the variable names used for MPI
- PR #3112⁶⁸⁰ - Minor fixes to `exclusive_scan` algorithm
- PR #3111⁶⁸¹ - Revert "fix detection of `cxx11_std_atomic`"
- PR #3110⁶⁸² - Suspend thread pool
- PR #3109⁶⁸³ - Fixing thread scheduling when yielding a thread id
- PR #3108⁶⁸⁴ - Revert "Suspend thread pool"
- PR #3107⁶⁸⁵ - Remove UB from `thread::id` relational operators
- PR #3106⁶⁸⁶ - Add cmake test for `std::decay_t` to fix cuda build
- PR #3105⁶⁸⁷ - Fixing refcount for async traversal frame
- PR #3104⁶⁸⁸ - Local execution of direct actions is now actually performed directly
- PR #3103⁶⁸⁹ - Adding support for generic `counter_raw_values` performance counter type
- Issue #3102⁶⁹⁰ - Introduce generic performance counter type returning an array of values
- PR #3101⁶⁹¹ - Revert "Adapting stack overhead limit for gcc 4.9"
- PR #3100⁶⁹² - Fix #3068 (`condition_variable` deadlock)

⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3122>

⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/3121>

⁶⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3120>

⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3119>

⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3118>

⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3117>

⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3116>

⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3115>

⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3114>

⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3113>

⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3112>

⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3111>

⁶⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3110>

⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3109>

⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3108>

⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3107>

⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3106>

⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3105>

⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3104>

⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3103>

⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3102>

⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3101>

⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3100>

- PR #3099⁶⁹³ - Fixing lock held during suspension in papi counter component
- PR #3098⁶⁹⁴ - Unbreak broadcast_wait_for_2822 test
- PR #3097⁶⁹⁵ - Adapting stack overhead limit for gcc 4.9
- PR #3096⁶⁹⁶ - fix detection of cxx11_std_atomic
- PR #3095⁶⁹⁷ - Add ciso646 header to get _LIBCPP_VERSION for testing inplace merge
- PR #3094⁶⁹⁸ - Relax atomic operations on performance counter values
- PR #3093⁶⁹⁹ - Short-circuit all_of/any_of/none_of instantiations
- PR #3092⁷⁰⁰ - Take advantage of C++14 lambda capture initialization syntax, where possible
- PR #3091⁷⁰¹ - Remove more references to Boost from logging code
- PR #3090⁷⁰² - Unify use of yield/yield_k
- PR #3089⁷⁰³ - Fix a strange thing in parallel::detail::handle_exception. (Fix #2834.)
- Issue #3088⁷⁰⁴ - A strange thing in parallel::sort.
- PR #3087⁷⁰⁵ - Fixing assertion in default_distribution_policy
- PR #3086⁷⁰⁶ - Implement parallel::remove and parallel::remove_if
- PR #3085⁷⁰⁷ - Addressing breaking changes in Boost V1.66
- PR #3084⁷⁰⁸ - Ignore build warnings round 2
- PR #3083⁷⁰⁹ - Fix typo HPX_WITH_MM_PREFECTH
- PR #3081⁷¹⁰ - Pre-decay template arguments early
- PR #3080⁷¹¹ - Suspend thread pool
- PR #3079⁷¹² - Ignore build warnings
- PR #3078⁷¹³ - Don't test inplace_merge with libc++
- PR #3076⁷¹⁴ - Fixing 3075: Part 1
- PR #3074⁷¹⁵ - Fix more build warnings

⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3099>

⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3098>

⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3097>

⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3096>

⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3095>

⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3094>

⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3093>

⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3092>

⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3091>

⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3090>

⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3089>

⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3088>

⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3087>

⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3086>

⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3085>

⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3084>

⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3083>

⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3081>

⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3080>

⁷¹² <https://github.com/STELLAR-GROUP/hpx/pull/3079>

⁷¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3078>

⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3076>

⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3074>

- [PR #3073](#)⁷¹⁶ - Suspend thread cleanup
- [PR #3072](#)⁷¹⁷ - Change existing `symbol_namespace::iterate` to return all data instead of invoking a callback
- [PR #3071](#)⁷¹⁸ - Fixing `pack_traversal_async` test
- [PR #3070](#)⁷¹⁹ - Fix `dynamic_counters_loaded_1508` test by adding dependency to `memory_component`
- [PR #3069](#)⁷²⁰ - Fix scheduling loop exit
- [Issue #3068](#)⁷²¹ - `hpx::lcos::condition_variable` could be suspect to deadlocks
- [PR #3067](#)⁷²² - `#ifdef` out `random_shuffle` deprecated in later c++
- [PR #3066](#)⁷²³ - Make coalescing test depend on coalescing library to ensure it gets built
- [PR #3065](#)⁷²⁴ - Workaround for `minimal_timed_async_executor_test` compilation failures, attempts to copy a deferred call (in unevaluated context)
- [PR #3064](#)⁷²⁵ - Fixing wrong condition in `wrapper_heap`
- [PR #3062](#)⁷²⁶ - Fix exception handling for `execution::seq`
- [PR #3061](#)⁷²⁷ - Adapt MSVC C++ mode handling to VS15.5
- [PR #3060](#)⁷²⁸ - Fix compiler problem in MSVC release mode
- [PR #3059](#)⁷²⁹ - Fixing #2931
- [Issue #3058](#)⁷³⁰ - `minimal_timed_async_executor_test_exe` fails to compile on master (d6f505c)
- [PR #3057](#)⁷³¹ - Fix `stable_merge_2964` compilation problems
- [PR #3056](#)⁷³² - Fix some build warnings caused by unused variables/unnecessary tests
- [PR #3055](#)⁷³³ - Update documentation for running tests
- [Issue #3054](#)⁷³⁴ - Assertion failure when using bulk `hpx::new_` in asynchronous mode
- [PR #3052](#)⁷³⁵ - Do not bind test running to `cmake` test build rule
- [PR #3051](#)⁷³⁶ - Fix HPX-Qt interaction in Qt example.
- [Issue #3048](#)⁷³⁷ - `nqueen` example fails occasionally
- [PR #3047](#)⁷³⁸ - Fixing #3044

⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3073>

⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3072>

⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3071>

⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3070>

⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3069>

⁷²¹ <https://github.com/STELLAR-GROUP/hpx/issues/3068>

⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/3067>

⁷²³ <https://github.com/STELLAR-GROUP/hpx/pull/3066>

⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3065>

⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3064>

⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3062>

⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3061>

⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3060>

⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3059>

⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3058>

⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3057>

⁷³² <https://github.com/STELLAR-GROUP/hpx/pull/3056>

⁷³³ <https://github.com/STELLAR-GROUP/hpx/pull/3055>

⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3054>

⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3052>

⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3051>

⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3048>

⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3047>

- PR #3046⁷³⁹ - Add OS thread suspension
- PR #3042⁷⁴⁰ - PyCicle - first attempt at a build tool for checking PR's
- PR #3041⁷⁴¹ - Fix a problem about asynchronous execution of `parallel::merge` and `parallel::partition`.
- PR #3040⁷⁴² - Fix a mistake about exception handling in asynchronous execution of `scan_partitioner`.
- PR #3039⁷⁴³ - Consistently use executors to schedule work
- PR #3038⁷⁴⁴ - Fixing local direct function execution and lambda actions perfect forwarding
- PR #3035⁷⁴⁵ - Make parallel unit test names match build target/folder names
- PR #3033⁷⁴⁶ - Fix setting of default build type
- Issue #3032⁷⁴⁷ - Fix partitioner arg copy found in #2982
- Issue #3031⁷⁴⁸ - Errors linking `libhpx.so` due to missing references (master branch, commit 6679a8882)
- PR #3030⁷⁴⁹ - Revert "implement executor then interface with `&&` forwarding reference"
- PR #3029⁷⁵⁰ - Run CI inspect checks before building
- PR #3028⁷⁵¹ - Added range version of `parallel::move`
- Issue #3027⁷⁵² - Implement all scheduling APIs in terms of executors
- PR #3026⁷⁵³ - implement executor then interface with `&&` forwarding reference
- PR #3025⁷⁵⁴ - Fix typo uninitialized to unitialized
- PR #3024⁷⁵⁵ - Inspect fixes
- PR #3023⁷⁵⁶ - P0356 Simplified partial function application
- PR #3022⁷⁵⁷ - Master fixes
- PR #3021⁷⁵⁸ - Segfault fix
- PR #3020⁷⁵⁹ - Disable command-line aliasing for applications that use `user_main`
- PR #3019⁷⁶⁰ - Adding `enable_elasticity` option to pool configuration
- PR #3018⁷⁶¹ - Fix stack overflow detection configuration in header files

⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3046>

⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3042>

⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3041>

⁷⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3040>

⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3039>

⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3038>

⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3035>

⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3033>

⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3032>

⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3031>

⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3030>

⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3029>

⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3028>

⁷⁵² <https://github.com/STELLAR-GROUP/hpx/issues/3027>

⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3026>

⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3025>

⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3024>

⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3023>

⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3022>

⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3021>

⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3020>

⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3019>

⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3018>

- PR #3017⁷⁶² - Speed up local action execution
- PR #3016⁷⁶³ - Unify stack-overflow detection options, remove reference to libsigsegv
- PR #3015⁷⁶⁴ - Speeding up accessing the resource partitioner and the topology info
- Issue #3014⁷⁶⁵ - HPX does not compile on POWER8 with gcc 5.4
- Issue #3013⁷⁶⁶ - hello_world occasionally prints multiple lines from a single OS-thread
- PR #3012⁷⁶⁷ - Silence warning about casting away qualifiers in itt_notify.hpp
- PR #3011⁷⁶⁸ - Fix cpuset leak in hwloc_topology_info.cpp
- PR #3010⁷⁶⁹ - Remove useless decay_copy
- PR #3009⁷⁷⁰ - Fixing 2996
- PR #3008⁷⁷¹ - Remove unused internal function
- PR #3007⁷⁷² - Fixing wrapper_heap alignment problems
- Issue #3006⁷⁷³ - hwloc memory leak
- PR #3004⁷⁷⁴ - Silence C4251 (needs to have dll-interface) for future_data_void
- Issue #3003⁷⁷⁵ - Suspension of runtime
- PR #3001⁷⁷⁶ - Attempting to avoid data races in async_traversal while evaluating dataflow()
- PR #3000⁷⁷⁷ - Adding hpx::util::optional as a first step to replace experimental::optional
- PR #2998⁷⁷⁸ - Cleanup up and Fixing component creation and deletion
- Issue #2996⁷⁷⁹ - Build fails with HPX_WITH_HWLOC=OFF
- PR #2995⁷⁸⁰ - Push more future_data functionality to source file
- PR #2994⁷⁸¹ - WIP: Fix throttle test
- PR #2993⁷⁸² - Making sure -hpx:help does not throw for required (but missing) arguments
- PR #2992⁷⁸³ - Adding non-blocking (on destruction) service executors
- Issue #2991⁷⁸⁴ - run_as_os_thread locks up

⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3017>

⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3016>

⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3015>

⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3014>

⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3013>

⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3012>

⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3011>

⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3010>

⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3009>

⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3008>

⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3007>

⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/3006>

⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3004>

⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3003>

⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3001>

⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3000>

⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2998>

⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2996>

⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2995>

⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2994>

⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2993>

⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2992>

⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2991>

- Issue #2990⁷⁸⁵ - `--help` will not work until all required options are provided
- PR #2989⁷⁸⁶ - Improve error messages caused by misuse of dataflow
- PR #2988⁷⁸⁷ - Improve error messages caused by misuse of `.then`
- Issue #2987⁷⁸⁸ - stack overflow detection producing false positives
- PR #2986⁷⁸⁹ - Deduplicate non-dependent `thread_info` logging types
- PR #2985⁷⁹⁰ - Adapted `parallel::{all_of|any_of|none_of}` for Ranges TS (see #1668)
- PR #2984⁷⁹¹ - Refactor `one_size_heap` code to simplify code
- PR #2983⁷⁹² - Fixing `local_new_component`
- PR #2982⁷⁹³ - Clang tidy
- PR #2981⁷⁹⁴ - Simplify allocator rebinding in pack traversal
- PR #2979⁷⁹⁵ - Fixing integer overflows
- PR #2978⁷⁹⁶ - Implement `parallel::inplace_merge`
- Issue #2977⁷⁹⁷ - Make `hwloc` compulsory instead of optional
- PR #2976⁷⁹⁸ - Making sure `client_base` instance that registered the component does not unregister it when being destructed
- PR #2975⁷⁹⁹ - Change version of pulled APEX to master
- PR #2974⁸⁰⁰ - Fix domain not being freed at the end of scheduling loop
- PR #2973⁸⁰¹ - Fix small typos
- PR #2972⁸⁰² - Adding `uintstd.h` header
- PR #2971⁸⁰³ - Fall back to creating local components using `local_new`
- PR #2970⁸⁰⁴ - Improve `is_tuple_like` trait
- PR #2969⁸⁰⁵ - Fix `HPX_WITH_MORE_THAN_64_THREADS` default value
- PR #2968⁸⁰⁶ - Cleaning up dataflow overload set
- PR #2967⁸⁰⁷ - Make `parallel::merge` is stable. (Fix #2964.)

⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2990>

⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2989>

⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2988>

⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2987>

⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2986>

⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2985>

⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2984>

⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2983>

⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2982>

⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2981>

⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2979>

⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2978>

⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2977>

⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2976>

⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2975>

⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2974>

⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2973>

⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2972>

⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2971>

⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2970>

⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2969>

⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2968>

⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2967>

- [PR #2966](#)⁸⁰⁸ - Fixing a couple of held locks during exception handling
- [PR #2965](#)⁸⁰⁹ - Adding missing `#include`
- [Issue #2964](#)⁸¹⁰ - parallel merge is not stable
- [PR #2963](#)⁸¹¹ - Making sure any function object passed to dataflow is released after being invoked
- [PR #2962](#)⁸¹² - Partially reverting #2891
- [PR #2961](#)⁸¹³ - Attempt to fix the gcc 4.9 problem with the async pack traversal
- [Issue #2959](#)⁸¹⁴ - Program terminates during error handling
- [Issue #2958](#)⁸¹⁵ - `HPX_PLAIN_ACTION` breaks due to missing include
- [PR #2957](#)⁸¹⁶ - Fixing errors generated by mixing different attribute syntaxes
- [Issue #2956](#)⁸¹⁷ - Mixing attribute syntaxes leads to compiler errors
- [Issue #2955](#)⁸¹⁸ - Fix OS-Thread throttling
- [PR #2953](#)⁸¹⁹ - Making sure any `hpx.os_threads=N` supplied through a `-hpx::config` file is taken into account
- [PR #2952](#)⁸²⁰ - Removing wrong call to `cleanup_terminated_locked`
- [PR #2951](#)⁸²¹ - Revert “Make sure the function vtables are initialized before use”
- [PR #2950](#)⁸²² - Fix a namespace compilation error when some schedulers are disabled
- [Issue #2949](#)⁸²³ - master branch giving lockups on shutdown
- [Issue #2947](#)⁸²⁴ - `hpx.ini` is not used correctly at initialization
- [PR #2946](#)⁸²⁵ - Adding explicit feature test for `thread_local`
- [PR #2945](#)⁸²⁶ - Make sure the function vtables are initialized before use
- [PR #2944](#)⁸²⁷ - Attempting to solve affinity problems on CircleCI
- [PR #2943](#)⁸²⁸ - Changing channel actions to be direct
- [PR #2942](#)⁸²⁹ - Adding `split_future` for `std::vector`
- [PR #2941](#)⁸³⁰ - Add a feature test to test for CXX11 override

⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2966>

⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2965>

⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2964>

⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2963>

⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/2962>

⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2961>

⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2959>

⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2958>

⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2957>

⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2956>

⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2955>

⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2953>

⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2952>

⁸²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2951>

⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/2950>

⁸²³ <https://github.com/STELLAR-GROUP/hpx/issues/2949>

⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2947>

⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2946>

⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2945>

⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2944>

⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2943>

⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2942>

⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2941>

- Issue #2940⁸³¹ - Add `split_future` for `future<vector<T>>`
- PR #2939⁸³² - Making error reporting during problems with setting affinity masks more verbose
- PR #2938⁸³³ - Fix this various executors
- PR #2937⁸³⁴ - Fix some typos in documentation
- PR #2934⁸³⁵ - Remove the need for “complete” SFINAE checks
- PR #2933⁸³⁶ - Making sure `parallel::for_loop` is executed in parallel if requested
- PR #2932⁸³⁷ - Classify `chunk_size_iterator` to input iterator tag. (Fix #2866)
- Issue #2931⁸³⁸ - `-hpx:help` triggers unusual error with clang build
- PR #2930⁸³⁹ - Add `#include` files needed to set `_POSIX_VERSION` for debug check
- PR #2929⁸⁴⁰ - Fix a couple of deprecated c++ features
- PR #2928⁸⁴¹ - Fixing execution parameters
- Issue #2927⁸⁴² - CMake warning: ... cycle in constraint graph
- PR #2926⁸⁴³ - Default pool rename
- Issue #2925⁸⁴⁴ - Default pool cannot be renamed
- Issue #2924⁸⁴⁵ - `hpx::attach-debugger=startup` does not work any more
- PR #2923⁸⁴⁶ - Alloc `membind`
- PR #2922⁸⁴⁷ - This fixes CircleCI errors when running with `-hpx:bind=none`
- PR #2921⁸⁴⁸ - Custom pool executor was missing priority and stacksize options
- PR #2920⁸⁴⁹ - Adding test to trigger problem reported in #2916
- PR #2919⁸⁵⁰ - Make sure the `resource_partitioner` is properly destructed on `hpx::finalize`
- Issue #2918⁸⁵¹ - `hpx::init` calls wrong (first) callback when called multiple times
- PR #2917⁸⁵² - Adding `util::checkpoint`
- Issue #2916⁸⁵³ - Weird runtime failures when using a channel and chained continuations

⁸³¹ <https://github.com/STELLAR-GROUP/hpx/issues/2940>

⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/2939>

⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/2938>

⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2937>

⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2934>

⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2933>

⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2932>

⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2931>

⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2930>

⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2929>

⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2928>

⁸⁴² <https://github.com/STELLAR-GROUP/hpx/issues/2927>

⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2926>

⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2925>

⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2924>

⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2923>

⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2922>

⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2921>

⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2920>

⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2919>

⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2918>

⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2917>

⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/2916>

- PR #2915⁸⁵⁴ - Introduce executor parameters customization points
- Issue #2914⁸⁵⁵ - Task assignment to current Pool has unintended consequences
- PR #2913⁸⁵⁶ - Fix rp hang
- PR #2912⁸⁵⁷ - Update contributors
- PR #2911⁸⁵⁸ - Fixing CUDA problems
- PR #2910⁸⁵⁹ - Improve error reporting for process component on POSIX systems
- PR #2909⁸⁶⁰ - Fix typo in include path
- PR #2908⁸⁶¹ - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2907⁸⁶² - Optionally force-delete remaining channel items on close
- PR #2906⁸⁶³ - Making sure generated performance counter names are correct
- Issue #2905⁸⁶⁴ - collecting idle-rate performance counters on multiple localities produces an error
- Issue #2904⁸⁶⁵ - build broken for Intel 17 compilers
- PR #2903⁸⁶⁶ - Documentation Updates– Adding New People
- PR #2902⁸⁶⁷ - Fixing service_executor
- PR #2901⁸⁶⁸ - Fixing partitioned_vector creation
- PR #2900⁸⁶⁹ - Add numa-balanced mode to hpx::bind, spread cores over numa domains
- Issue #2899⁸⁷⁰ - hpx::bind does not have a mode that balances cores over numa domains
- PR #2898⁸⁷¹ - Adding missing #include and missing guard for optional code section
- PR #2897⁸⁷² - Removing dependency on Boost.ICL
- Issue #2896⁸⁷³ - Debug build fails without -fpermissive with GCC 7.1 and Boost 1.65
- PR #2895⁸⁷⁴ - Fixing SLURM environment parsing
- PR #2894⁸⁷⁵ - Fix incorrect handling of compile definition with value 0
- Issue #2893⁸⁷⁶ - Disabling schedulers causes build errors

⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2915>

⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2914>

⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2913>

⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2912>

⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2911>

⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2910>

⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2909>

⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2908>

⁸⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2907>

⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2906>

⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2905>

⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2904>

⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2903>

⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2902>

⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2901>

⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2900>

⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2899>

⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2898>

⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2897>

⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/2896>

⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2895>

⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2894>

⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2893>

- PR #2892⁸⁷⁷ - added list serializer
- PR #2891⁸⁷⁸ - Resource Partitioner Fixes
- Issue #2890⁸⁷⁹ - Destroying a non-empty channel causes an assertion failure
- PR #2889⁸⁸⁰ - Add check for libatomic
- PR #2888⁸⁸¹ - Fix compilation problems if HPX_WITH_ITT_NOTIFY=ON
- PR #2887⁸⁸² - Adapt broadcast() to non-unwrapping async<Action>
- PR #2886⁸⁸³ - Replace Boost.Random with C++11 <random>
- Issue #2885⁸⁸⁴ - regression in broadcast?
- Issue #2884⁸⁸⁵ - linking -latomic is not portable
- PR #2883⁸⁸⁶ - Explicitly set -pthread flag if available
- PR #2882⁸⁸⁷ - Wrap boost::format uses
- Issue #2881⁸⁸⁸ - hpx not compiling with HPX_WITH_ITT_NOTIFY=On
- Issue #2880⁸⁸⁹ - hpx::bind scatter/balanced give wrong pu masks
- PR #2878⁸⁹⁰ - Fix incorrect pool usage masks setup in RP/thread manager
- PR #2877⁸⁹¹ - Require std::array by default
- PR #2875⁸⁹² - Deprecate use of BOOST_ASSERT
- PR #2874⁸⁹³ - Changed serialization of boost.variant to use variadic templates
- Issue #2873⁸⁹⁴ - building with parcellport_mpi fails on cori
- PR #2871⁸⁹⁵ - Adding missing support for throttling scheduler
- PR #2870⁸⁹⁶ - Disambiguate use of base_lco_with_value macros with channel
- Issue #2869⁸⁹⁷ - Difficulty compiling HPX_REGISTER_CHANNEL_DECLARATION(double)
- PR #2868⁸⁹⁸ - Removing unneeded assert
- PR #2867⁸⁹⁹ - Implement parallel::unique

⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2892>

⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2891>

⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2890>

⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2889>

⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2888>

⁸⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2887>

⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2886>

⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2885>

⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2884>

⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2883>

⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2882>

⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2881>

⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2880>

⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2878>

⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2877>

⁸⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2875>

⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2874>

⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2873>

⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2871>

⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2870>

⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2869>

⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2868>

⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2867>

- [Issue #2866](#)⁹⁰⁰ - The `chunk_size_iterator` violates multipass guarantee
- [PR #2865](#)⁹⁰¹ - Only use `sched_getcpu` on linux machines
- [PR #2864](#)⁹⁰² - Create redistribution archive for successful builds
- [PR #2863](#)⁹⁰³ - Replace casts/assignments with hard-coded `memcpy` operations
- [Issue #2862](#)⁹⁰⁴ - `sched_getcpu` not available on MacOS
- [PR #2861](#)⁹⁰⁵ - Fixing unmatched header defines and recursive inclusion of `threadmanager`
- [Issue #2860](#)⁹⁰⁶ - Master program fails with assertion '`type == data_type_address`' failed: `HPX(assertion_failure)`
- [Issue #2852](#)⁹⁰⁷ - Support for ARM64
- [PR #2858](#)⁹⁰⁸ - Fix misplaced `#if #endif`'s that cause build failure without `THREAD_CUMULATIVE_COUNTS`
- [PR #2857](#)⁹⁰⁹ - Fix some listing in documentation
- [PR #2856](#)⁹¹⁰ - Fixing component handling for `lcoss`
- [PR #2855](#)⁹¹¹ - Add documentation for `coarrays`
- [PR #2854](#)⁹¹² - Support ARM64 in timestamps
- [PR #2853](#)⁹¹³ - Update Table 17. Non-modifying Parallel Algorithms in Documentation
- [PR #2851](#)⁹¹⁴ - Allowing for non-default-constructible component types
- [PR #2850](#)⁹¹⁵ - Enable returning `future<R>` from actions where `R` is not default-constructible
- [PR #2849](#)⁹¹⁶ - Unify serialization of non-default-constructable types
- [Issue #2848](#)⁹¹⁷ - Components have to be default constructible
- [Issue #2847](#)⁹¹⁸ - Returning a `future<R>` where `R` is not default-constructable broken
- [Issue #2846](#)⁹¹⁹ - Unify serialization of non-default-constructible types
- [PR #2845](#)⁹²⁰ - Add Visual Studio 2015 to the tested toolchains in Appveyor
- [Issue #2844](#)⁹²¹ - Change the appveyor build to use the minimal required MSVC version
- [Issue #2843](#)⁹²² - multi node `hello_world` hangs

⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2866>

⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2865>

⁹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2864>

⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2863>

⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2862>

⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2861>

⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2860>

⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2852>

⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2858>

⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2857>

⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2856>

⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2855>

⁹¹² <https://github.com/STELLAR-GROUP/hpx/pull/2854>

⁹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2853>

⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2851>

⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2850>

⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2849>

⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2848>

⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2847>

⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2846>

⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2845>

⁹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/2844>

⁹²² <https://github.com/STELLAR-GROUP/hpx/issues/2843>

- PR #2842⁹²³ - Correcting Spelling mistake in docs
- PR #2841⁹²⁴ - Fix usage of `std::aligned_storage`
- PR #2840⁹²⁵ - Remove `constexpr` from a void function
- Issue #2839⁹²⁶ - `memcpy` buffer overflow: `load_construct_data()` and `std::complex` members
- Issue #2835⁹²⁷ - `constexpr` functions with `void` return type break compilation with CUDA 8.0
- Issue #2834⁹²⁸ - One suspicion in `parallel::detail::handle_exception`
- PR #2833⁹²⁹ - Implement `parallel::merge`
- PR #2832⁹³⁰ - Fix a strange thing in `parallel::util::detail::handle_local_exceptions`. (Fix #2818)
- PR #2830⁹³¹ - Break the debugger when a test failed
- Issue #2831⁹³² - `parallel/executors/execution_fwd.hpp` causes compilation failure in C++11 mode.
- PR #2829⁹³³ - Implement an API for asynchronous pack traversal
- PR #2828⁹³⁴ - Split unit test builds on CircleCI to avoid timeouts
- Issue #2827⁹³⁵ - failure to compile `hello_world` example with `-Werror`
- PR #2824⁹³⁶ - Making sure promises are marked as started when used as continuations
- PR #2823⁹³⁷ - Add documentation for `partitioned_vector_view`
- Issue #2822⁹³⁸ - Yet another issue with `wait_for` similar to #2796
- PR #2821⁹³⁹ - Fix bugs and improve that about `HPX_HAVE_CXX11_AUTO_RETURN_VALUE` of CMake
- PR #2820⁹⁴⁰ - Support C++11 in benchmark codes of `parallel::partition` and `parallel::partition_copy`
- PR #2819⁹⁴¹ - Fix compile errors in unit test of container version of `parallel::partition`
- Issue #2818⁹⁴² - A strange thing in `parallel::util::detail::handle_local_exceptions`
- Issue #2815⁹⁴³ - HPX fails to compile with `HPX_WITH_CUDA=ON` and the new CUDA 9.0 RC
- Issue #2814⁹⁴⁴ - Using 'gmakeN' after 'cmake' produces error in `src/CMakeFiles/hpx.dir/runtime/agas/addressing_service.cpp.o`

⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/2842>

⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2841>

⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2840>

⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2839>

⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2835>

⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2834>

⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2833>

⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2832>

⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2830>

⁹³² <https://github.com/STELLAR-GROUP/hpx/issues/2831>

⁹³³ <https://github.com/STELLAR-GROUP/hpx/pull/2829>

⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2828>

⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2827>

⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2824>

⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2823>

⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2822>

⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2821>

⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2820>

⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2819>

⁹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/2818>

⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2815>

⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2814>

- PR #2813⁹⁴⁵ - Properly support `[[noreturn]]` attribute if available
- Issue #2812⁹⁴⁶ - Compilation fails with gcc 7.1.1
- PR #2811⁹⁴⁷ - Adding `hpx::launch::lazy` and support for `async`, `dataflow`, and `future::then`
- PR #2810⁹⁴⁸ - Add option allowing to disable deprecation warning
- PR #2809⁹⁴⁹ - Disable throttling scheduler if HWLOC is not found/used
- PR #2808⁹⁵⁰ - Fix compile errors on some environments of `parallel::partition`
- Issue #2807⁹⁵¹ - Difficulty building with `HPX_WITH_HWLOC=Off`
- PR #2806⁹⁵² - Partitioned vector
- PR #2805⁹⁵³ - Serializing collections with non-default constructible data
- PR #2802⁹⁵⁴ - Fix FreeBSD 11
- Issue #2801⁹⁵⁵ - Rate limiting techniques in `io_service`
- Issue #2800⁹⁵⁶ - New Launch Policy: `async_if`
- PR #2799⁹⁵⁷ - Fix a unit test failure on GCC in `tuple_cat`
- PR #2798⁹⁵⁸ - bump minimum required cmake to 3.0 in test
- PR #2797⁹⁵⁹ - Making sure `future::wait_for` et.al. work properly for action results
- Issue #2796⁹⁶⁰ - `wait_for` does always in “deferred” state for calls on remote localities
- Issue #2795⁹⁶¹ - Serialization of types without default constructor
- PR #2794⁹⁶² - Fixing test for `partitioned_vector` iteration
- PR #2792⁹⁶³ - Implemented segmented find and its variations for partitioned vector
- PR #2791⁹⁶⁴ - Circumvent scary warning about placement new
- PR #2790⁹⁶⁵ - Fix OSX build
- PR #2789⁹⁶⁶ - Resource partitioner
- PR #2788⁹⁶⁷ - Adapt `parallel::is_heap` and `parallel::is_heap_until` to Ranges TS

⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2813>

⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2812>

⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2811>

⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2810>

⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2809>

⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2808>

⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2807>

⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2806>

⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2805>

⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2802>

⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2801>

⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2800>

⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2799>

⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2798>

⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2797>

⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2796>

⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2795>

⁹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2794>

⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2792>

⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2791>

⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2790>

⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2789>

⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2788>

- PR #2787⁹⁶⁸ - Unwrap hotfixes
- PR #2786⁹⁶⁹ - Update CMake Minimum Version to 3.3.2 (refs #2565)
- Issue #2785⁹⁷⁰ - Issues with masks and cpuset
- PR #2784⁹⁷¹ - Error with reduce and transform reduce fixed
- PR #2783⁹⁷² - StackOverflow integration with libsigsegv
- PR #2782⁹⁷³ - Replace boost::atomic with std::atomic (where possible)
- PR #2781⁹⁷⁴ - Check for and optionally use [[deprecated]] attribute
- PR #2780⁹⁷⁵ - Adding empty (but non-trivial) destructor to circumvent warnings
- PR #2779⁹⁷⁶ - Exception info tweaks
- PR #2778⁹⁷⁷ - Implement parallel::partition
- PR #2777⁹⁷⁸ - Improve error handling in gather_here/gather_there
- PR #2776⁹⁷⁹ - Fix a bug in compiler version check
- PR #2775⁹⁸⁰ - Fix compilation when HPX_WITH_LOGGING is OFF
- PR #2774⁹⁸¹ - Removing dependency on Boost.Date_Time
- PR #2773⁹⁸² - Add sync_images() method to spmd_block class
- PR #2772⁹⁸³ - Adding documentation for PAPI counters
- PR #2771⁹⁸⁴ - Removing boost preprocessor dependency
- PR #2770⁹⁸⁵ - Adding test, fixing deadlock in config registry
- PR #2769⁹⁸⁶ - Remove some other warnings and errors detected by clang 5.0
- Issue #2768⁹⁸⁷ - Is there iterator tag for HPX?
- PR #2767⁹⁸⁸ - Improvements to continuation annotation
- PR #2765⁹⁸⁹ - gcc split stack support for HPX threads #620
- PR #2764⁹⁹⁰ - Fix some uses of begin/end, remove unnecessary includes

⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2787>

⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2786>

⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2785>

⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2784>

⁹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2783>

⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2782>

⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2781>

⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2780>

⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2779>

⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2778>

⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2777>

⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2776>

⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2775>

⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2774>

⁹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2773>

⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2772>

⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2771>

⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2770>

⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2769>

⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2768>

⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2767>

⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2765>

⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2764>

- [PR #2763](#)⁹⁹¹ - Bump minimal Boost version to 1.55.0
- [PR #2762](#)⁹⁹² - `hpx::partitioned_vector` serializer
- [PR #2761](#)⁹⁹³ - Adding configuration summary to cmake output and `-hpx:info`
- [PR #2760](#)⁹⁹⁴ - Removing `1d_hydro` example as it is broken
- [PR #2758](#)⁹⁹⁵ - Remove various warnings detected by clang 5.0
- [Issue #2757](#)⁹⁹⁶ - In case of a “raw thread” is needed per core for implementing parallel algorithm, what is good practice in HPX?
- [PR #2756](#)⁹⁹⁷ - Allowing for LCOs to be simple components
- [PR #2755](#)⁹⁹⁸ - Removing `make_index_pack_unrolled`
- [PR #2754](#)⁹⁹⁹ - Implement `parallel::unique_copy`
- [PR #2753](#)¹⁰⁰⁰ - Fixing detection of `[[fallthrough]]` attribute
- [PR #2752](#)¹⁰⁰¹ - New thread priority names
- [PR #2751](#)¹⁰⁰² - Replace `boost::exception` with proposed `exception_info`
- [PR #2750](#)¹⁰⁰³ - Replace `boost::iterator_range`
- [PR #2749](#)¹⁰⁰⁴ - Fixing hdf5 examples
- [Issue #2748](#)¹⁰⁰⁵ - HPX fails to build with enabled hdf5 examples
- [Issue #2747](#)¹⁰⁰⁶ - Inherited task priorities break certain DAG optimizations
- [Issue #2746](#)¹⁰⁰⁷ - HPX segfaulting with valgrind
- [PR #2745](#)¹⁰⁰⁸ - Adding extended arithmetic performance counters
- [PR #2744](#)¹⁰⁰⁹ - Adding ability to statistics counters to reset base counter
- [Issue #2743](#)¹⁰¹⁰ - Statistics counter does not support resetting
- [PR #2742](#)¹⁰¹¹ - Making sure Vc V2 builds without additional HPX configuration flags
- [PR #2741](#)¹⁰¹² - Deprecate `unwrapped` and implement `unwrap` and `unwrapping`
- [PR #2740](#)¹⁰¹³ - Coroutine stackoverflow detection for linux/posix; [Issue #2408](#)

⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2763>

⁹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2762>

⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2761>

⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2760>

⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2758>

⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2757>

⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2756>

⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2755>

⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2754>

¹⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2753>

¹⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2752>

¹⁰⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2751>

¹⁰⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2750>

¹⁰⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2749>

¹⁰⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2748>

¹⁰⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2747>

¹⁰⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2746>

¹⁰⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2745>

¹⁰⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2744>

¹⁰¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2743>

¹⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2742>

¹⁰¹² <https://github.com/STELLAR-GROUP/hpx/pull/2741>

¹⁰¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2740>

- PR #2739¹⁰¹⁴ - Add files via upload
- PR #2738¹⁰¹⁵ - Appveyor support
- PR #2737¹⁰¹⁶ - Fixing 2735
- Issue #2736¹⁰¹⁷ - 1d_hydro example doesn't work
- Issue #2735¹⁰¹⁸ - partitioned_vector_subview test failing
- PR #2734¹⁰¹⁹ - Add C++11 range utilities
- PR #2733¹⁰²⁰ - Adapting iterator requirements for parallel algorithms
- PR #2732¹⁰²¹ - Integrate C++ Co-arrays
- PR #2731¹⁰²² - Adding on_migrated event handler to migratable component instances
- Issue #2729¹⁰²³ - Add on_migrated() event handler to migratable components
- Issue #2728¹⁰²⁴ - Why Projection is needed in parallel algorithms?
- PR #2727¹⁰²⁵ - Cmake files for StackOverflow Detection
- PR #2726¹⁰²⁶ - CMake for Stack Overflow Detection
- PR #2725¹⁰²⁷ - Implemented segmented algorithms for partitioned vector
- PR #2724¹⁰²⁸ - Fix examples in Action documentation
- PR #2723¹⁰²⁹ - Enable lcos::channel<T>::register_as
- Issue #2722¹⁰³⁰ - channel register_as() failing on compilation
- PR #2721¹⁰³¹ - Mind map
- PR #2720¹⁰³² - reorder forward declarations to get rid of C++14-only auto return types
- PR #2719¹⁰³³ - Add documentation for partitioned_vector and add features in pack.hpp
- Issue #2718¹⁰³⁴ - Some forward declarations in execution_fwd.hpp aren't C++11-compatible
- PR #2717¹⁰³⁵ - Config support for fallthrough attribute
- PR #2716¹⁰³⁶ - Implement parallel::partition_copy

¹⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2739>

¹⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2738>

¹⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2737>

¹⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2736>

¹⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2735>

¹⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2734>

¹⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2733>

¹⁰²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2732>

¹⁰²² <https://github.com/STELLAR-GROUP/hpx/pull/2731>

¹⁰²³ <https://github.com/STELLAR-GROUP/hpx/issues/2729>

¹⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2728>

¹⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2727>

¹⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2726>

¹⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2725>

¹⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2724>

¹⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2723>

¹⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2722>

¹⁰³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2721>

¹⁰³² <https://github.com/STELLAR-GROUP/hpx/pull/2720>

¹⁰³³ <https://github.com/STELLAR-GROUP/hpx/pull/2719>

¹⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2718>

¹⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2717>

¹⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2716>

- PR #2715¹⁰³⁷ - initial import of icu string serializer
- PR #2714¹⁰³⁸ - initial import of valarray serializer
- PR #2713¹⁰³⁹ - Remove slashes before CMAKE_FILES_DIRECTORY variables
- PR #2712¹⁰⁴⁰ - Fixing wait for 1751
- PR #2711¹⁰⁴¹ - Adjust code for minimal supported GCC having being bumped to 4.9
- PR #2710¹⁰⁴² - Adding code of conduct
- PR #2709¹⁰⁴³ - Fixing UB in destroy tests
- PR #2708¹⁰⁴⁴ - Add inline to prevent multiple definition issue
- Issue #2707¹⁰⁴⁵ - Multiple defined symbols for task_block.hpp in VS2015
- PR #2706¹⁰⁴⁶ - Adding .clang-format file
- PR #2704¹⁰⁴⁷ - Add a synchronous mapping API
- Issue #2703¹⁰⁴⁸ - Request: Add the .clang-format file to the repository
- Issue #2702¹⁰⁴⁹ - STELLAR-GROUP/Vc slower than VCv1 possibly due to wrong instructions generated
- Issue #2701¹⁰⁵⁰ - Datapar with STELLAR-GROUP/Vc requires obscure flag
- Issue #2700¹⁰⁵¹ - Naming inconsistency in parallel algorithms
- Issue #2699¹⁰⁵² - Iterator requirements are different from standard in parallel copy_if.
- PR #2698¹⁰⁵³ - Properly releasing parcellport write handlers
- Issue #2697¹⁰⁵⁴ - Compile error in addressing_service.cpp
- Issue #2696¹⁰⁵⁵ - Building and using HPX statically: undefined references from runtime_support_server.cpp
- Issue #2695¹⁰⁵⁶ - Executor changes cause compilation failures
- PR #2694¹⁰⁵⁷ - Refining C++ language mode detection for MSVC
- PR #2693¹⁰⁵⁸ - P0443 r2
- PR #2692¹⁰⁵⁹ - Partially reverting changes to parcel_await

¹⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2715>

¹⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2714>

¹⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2713>

¹⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2712>

¹⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2711>

¹⁰⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2710>

¹⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2709>

¹⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2708>

¹⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2707>

¹⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2706>

¹⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2704>

¹⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2703>

¹⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2702>

¹⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2701>

¹⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2700>

¹⁰⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2699>

¹⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2698>

¹⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2697>

¹⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2696>

¹⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2695>

¹⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2694>

¹⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2693>

¹⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2692>

- [Issue #2689](#)¹⁰⁶⁰ - HPX build fails when HPX_WITH_CUDA is enabled
- [PR #2688](#)¹⁰⁶¹ - Make Cuda Clang builds pass
- [PR #2687](#)¹⁰⁶² - Add an `is_tuple_like` trait for sequenceable type detection
- [PR #2686](#)¹⁰⁶³ - Allowing throttling scheduler to be used without idle backoff
- [PR #2685](#)¹⁰⁶⁴ - Add support of `std::array` to `hpx::util::tuple_size` and `tuple_element`
- [PR #2684](#)¹⁰⁶⁵ - Adding new statistics performance counters
- [PR #2683](#)¹⁰⁶⁶ - Replace `boost::exception_ptr` with `std::exception_ptr`
- [Issue #2682](#)¹⁰⁶⁷ - HPX does not compile with HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF
- [PR #2681](#)¹⁰⁶⁸ - Attempt to fix problem in `managed_component_base`
- [PR #2680](#)¹⁰⁶⁹ - Fix bad size during archive creation
- [Issue #2679](#)¹⁰⁷⁰ - Mismatch between size of archive and container
- [Issue #2678](#)¹⁰⁷¹ - In parallel algorithm, other tasks are executed to the end even if an exception occurs in any task.
- [PR #2677](#)¹⁰⁷² - Adding include check for `std::addressof`
- [PR #2676](#)¹⁰⁷³ - Adding `parallel::destroy` and `destroy_n`
- [PR #2675](#)¹⁰⁷⁴ - Making sure statistics counters work as expected
- [PR #2674](#)¹⁰⁷⁵ - Turning assertions into exceptions
- [PR #2673](#)¹⁰⁷⁶ - Inhibit direct conversion from `future<future<T>> -> future<void>`
- [PR #2672](#)¹⁰⁷⁷ - C++17 invoke forms
- [PR #2671](#)¹⁰⁷⁸ - Adding `uninitialized_value_construct` and `uninitialized_value_construct_n`
- [PR #2670](#)¹⁰⁷⁹ - Integrate `spmd` multidimensionnal views for `partitioned_vectors`
- [PR #2669](#)¹⁰⁸⁰ - Adding `uninitialized_default_construct` and `uninitialized_default_construct_n`
- [PR #2668](#)¹⁰⁸¹ - Fixing documentation index
- [Issue #2667](#)¹⁰⁸² - Ambiguity of nested `hpx::future<void>`'s

¹⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2689>

¹⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2688>

¹⁰⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2687>

¹⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2686>

¹⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2685>

¹⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2684>

¹⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2683>

¹⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2682>

¹⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2681>

¹⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2680>

¹⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2679>

¹⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/2678>

¹⁰⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2677>

¹⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2676>

¹⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2675>

¹⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2674>

¹⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2673>

¹⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2672>

¹⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2671>

¹⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2670>

¹⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2669>

¹⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2668>

¹⁰⁸² <https://github.com/STELLAR-GROUP/hpx/issues/2667>

- Issue #2666¹⁰⁸³ - Statistics Performance counter is not working
- PR #2664¹⁰⁸⁴ - Adding uninitialized_move and uninitialized_move_n
- Issue #2663¹⁰⁸⁵ - Seg fault in managed_component::get_base_gid, possibly cause by util::reinitializable_static
- Issue #2662¹⁰⁸⁶ - Crash in managed_component::get_base_gid due to problem with util::reinitializable_static
- PR #2665¹⁰⁸⁷ - Hide the detail namespace in doxygen per default
- PR #2660¹⁰⁸⁸ - Add documentation to hpx::util::unwrapped and hpx::util::unwrapped2
- PR #2659¹⁰⁸⁹ - Improve integration with vcpkg
- PR #2658¹⁰⁹⁰ - Unify access_data trait for use in both, serialization and de-serialization
- PR #2657¹⁰⁹¹ - Removing hpx::lcos::queue<T>
- PR #2656¹⁰⁹² - Reduce MAX_TERMINATED_THREADS default, improve memory use on manycore cpus
- PR #2655¹⁰⁹³ - Maintenance for emulate-deleted macros
- PR #2654¹⁰⁹⁴ - Implement parallel is_heap and is_heap_until
- PR #2653¹⁰⁹⁵ - Drop support for VS2013
- PR #2652¹⁰⁹⁶ - This patch makes sure that all parcels in a batch are properly handled
- PR #2649¹⁰⁹⁷ - Update docs (Table 18) - move transform to end
- Issue #2647¹⁰⁹⁸ - hpx::parcelset::detail::parcel_data::has_continuation_ is uninitialized
- Issue #2644¹⁰⁹⁹ - Some .vcxproj in the HPX.sln fail to build
- Issue #2641¹¹⁰⁰ - hpx::lcos::queue should be deprecated
- PR #2640¹¹⁰¹ - A new throttling policy with public APIs to suspend/resume
- PR #2639¹¹⁰² - Fix a tiny typo in tutorial.
- Issue #2638¹¹⁰³ - Invalid return type 'void' of constexpr function
- PR #2636¹¹⁰⁴ - Add and use HPX_MSVC_WARNING_PRAGMA for #pragma warning
- PR #2633¹¹⁰⁵ - Distributed define_spmd_block

¹⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/2666>

¹⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2664>

¹⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2663>

¹⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2662>

¹⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2665>

¹⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2660>

¹⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2659>

¹⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2658>

¹⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2657>

¹⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2656>

¹⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2655>

¹⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2654>

¹⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2653>

¹⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2652>

¹⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2649>

¹⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2647>

¹⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2644>

¹¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2641>

¹¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2640>

¹¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2639>

¹¹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2638>

¹¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2636>

¹¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2633>

- PR #2632¹¹⁰⁶ - Making sure container serialization uses size-compatible types
- PR #2631¹¹⁰⁷ - Add `lcos::local::one_element_channel`
- PR #2629¹¹⁰⁸ - Move `unordered_map` out of `parcelport` into `hpx/concurrent`
- PR #2628¹¹⁰⁹ - Making sure that shutdown does not hang
- PR #2627¹¹¹⁰ - Fix serialization
- PR #2626¹¹¹¹ - Generate `cmake_variables.qbk` and `cmake_toolchains.qbk` outside of the source tree
- PR #2625¹¹¹² - Supporting `-std=c++17` flag
- PR #2624¹¹¹³ - Fixing a small `cmake` typo
- PR #2622¹¹¹⁴ - Update CMake minimum required version to 3.0.2 (closes #2621)
- Issue #2621¹¹¹⁵ - Compiling `hpx` master fails with `/usr/bin/ld: final link failed: Bad value`
- PR #2620¹¹¹⁶ - Remove warnings due to some captured variables
- PR #2619¹¹¹⁷ - LF multiple parcels
- PR #2618¹¹¹⁸ - Some fixes to `libfabric` that didn't get caught before the merge
- PR #2617¹¹¹⁹ - Adding `hpx::local_new`
- PR #2616¹¹²⁰ - Documentation: Extract all entities in order to autolink functions correctly
- Issue #2615¹¹²¹ - Documentation: Linking functions is broken
- PR #2614¹¹²² - Adding serialization for `std::deque`
- PR #2613¹¹²³ - We need to link with `boost.thread` and `boost.chrono` if we use `boost.context`
- PR #2612¹¹²⁴ - Making sure `for_loop_n(par, ...)` is actually executed in parallel
- PR #2611¹¹²⁵ - Add documentation to `invoke_fused` and friends NFC
- PR #2610¹¹²⁶ - Added reduction templates using an identity value
- PR #2608¹¹²⁷ - Fixing some unused vars in `inspect`
- PR #2607¹¹²⁸ - Fixed build for mingw

¹¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2632>
¹¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2631>
¹¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2629>
¹¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2628>
¹¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2627>
¹¹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2626>
¹¹¹² <https://github.com/STELLAR-GROUP/hpx/pull/2625>
¹¹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2624>
¹¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2622>
¹¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2621>
¹¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2620>
¹¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2619>
¹¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2618>
¹¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2617>
¹¹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2616>
¹¹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/2615>
¹¹²² <https://github.com/STELLAR-GROUP/hpx/pull/2614>
¹¹²³ <https://github.com/STELLAR-GROUP/hpx/pull/2613>
¹¹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2612>
¹¹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2611>
¹¹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2610>
¹¹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2608>
¹¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2607>

- PR #2606¹¹²⁹ - Supporting generic context for boost >= 1.61
- PR #2605¹¹³⁰ - Parcelport libfabric3
- PR #2604¹¹³¹ - Adding allocator support to promise and friends
- PR #2603¹¹³² - Barrier hang
- PR #2602¹¹³³ - Changes to scheduler to steal from one high-priority queue
- Issue #2601¹¹³⁴ - High priority tasks are not executed first
- PR #2600¹¹³⁵ - Compat fixes
- PR #2599¹¹³⁶ - Compatibility layer for threading support
- PR #2598¹¹³⁷ - V1.1
- PR #2597¹¹³⁸ - Release V1.0
- PR #2592¹¹³⁹ - First attempt to introduce spmd_block in hpx
- PR #2586¹¹⁴⁰ - local_segment in segmented_iterator_traits
- Issue #2584¹¹⁴¹ - Add allocator support to promise, packaged_task and friends
- PR #2576¹¹⁴² - Add missing dependencies of cuda based tests
- PR #2575¹¹⁴³ - Remove warnings due to some captured variables
- Issue #2574¹¹⁴⁴ - MSVC 2015 Compiler crash when building HPX
- Issue #2568¹¹⁴⁵ - Remove throttle_scheduler as it has been abandoned
- Issue #2566¹¹⁴⁶ - Add an inline versioning namespace before 1.0 release
- Issue #2565¹¹⁴⁷ - Raise minimal cmake version requirement
- PR #2556¹¹⁴⁸ - Fixing scan partitioner
- PR #2546¹¹⁴⁹ - Broadcast async
- Issue #2543¹¹⁵⁰ - make install fails due to a non-existing .so file
- PR #2495¹¹⁵¹ - wait_or_add_new returning thread_id_type

¹¹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2606>

¹¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2605>

¹¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2604>

¹¹³² <https://github.com/STELLAR-GROUP/hpx/pull/2603>

¹¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/2602>

¹¹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2601>

¹¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2600>

¹¹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2599>

¹¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2598>

¹¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2597>

¹¹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2592>

¹¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2586>

¹¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/2584>

¹¹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2576>

¹¹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2575>

¹¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2574>

¹¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2568>

¹¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2566>

¹¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2565>

¹¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2556>

¹¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2546>

¹¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2543>

¹¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2495>

- Issue #2480¹¹⁵² - Unable to register new performance counter
- Issue #2471¹¹⁵³ - no type named 'fcontext_t' in namespace
- Issue #2456¹¹⁵⁴ - Re-implement hpx::util::unwrapped
- Issue #2455¹¹⁵⁵ - Add more arithmetic performance counters
- PR #2454¹¹⁵⁶ - Fix a couple of warnings and compiler errors
- PR #2453¹¹⁵⁷ - Timed executor support
- PR #2447¹¹⁵⁸ - Implementing new executor API (P0443)
- Issue #2439¹¹⁵⁹ - Implement executor proposal
- Issue #2408¹¹⁶⁰ - Stackoverflow detection for linux, e.g. based on libsigsegv
- PR #2377¹¹⁶¹ - Add a customization point for put_parcel so we can override actions
- Issue #2368¹¹⁶² - HPX_ASSERT problem
- Issue #2324¹¹⁶³ - Change default number of threads used to the maximum of the system
- Issue #2266¹¹⁶⁴ - hpx_0.9.99 make tests fail
- PR #2195¹¹⁶⁵ - Support for code completion in VIM
- Issue #2137¹¹⁶⁶ - Hpx does not compile over osx
- Issue #2092¹¹⁶⁷ - make tests should just build the tests
- Issue #2026¹¹⁶⁸ - Build HPX with Apple's clang
- Issue #1932¹¹⁶⁹ - hpx with PBS fails on multiple localities
- PR #1914¹¹⁷⁰ - Parallel heap algorithm implementations WIP
- Issue #1598¹¹⁷¹ - Disconnecting a locality results in segfault using heartbeat example
- Issue #1404¹¹⁷² - unwrapped doesn't work with movable only types
- Issue #1400¹¹⁷³ - hpx::util::unwrapped doesn't work with non-future types
- Issue #1205¹¹⁷⁴ - TSS is broken

¹¹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2480>

¹¹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/2471>

¹¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2456>

¹¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2455>

¹¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2454>

¹¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2453>

¹¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2447>

¹¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2439>

¹¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2408>

¹¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2377>

¹¹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2368>

¹¹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/2324>

¹¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2266>

¹¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2195>

¹¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2137>

¹¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2092>

¹¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2026>

¹¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1932>

¹¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1914>

¹¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1598>

¹¹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1404>

¹¹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1400>

¹¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1205>

- [Issue #1126](#)¹¹⁷⁵ - `vector<future<T>>` does not work gracefully with dataflow, `when_all` and `unwrapped`
- [Issue #1056](#)¹¹⁷⁶ - Thread manager cleanup
- [Issue #863](#)¹¹⁷⁷ - Futures should not require a default constructor
- [Issue #856](#)¹¹⁷⁸ - Allow `runtime_mode_connect` to be used with security enabled
- [Issue #726](#)¹¹⁷⁹ - Valgrind
- [Issue #701](#)¹¹⁸⁰ - Add RCR performance counter component
- [Issue #528](#)¹¹⁸¹ - 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](#)¹¹⁸² 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](#)¹¹⁸³. Please see [Issue #2333](#)¹¹⁸⁴ 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](#)¹¹⁸⁵ for more information.
- The parameter sequence for the `hpx::parallel::transform_reduce` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17. The old overload can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.
- The algorithm `hpx::parallel::inner_product` has been renamed to `hpx::parallel::transform_reduce` to match the changes this algorithm has undergone while being moved to C++17. The old `inner_product` names can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.

¹¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1126>

¹¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1056>

¹¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/863>

¹¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/856>

¹¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/726>

¹¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/701>

¹¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/528>

¹¹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2289>

¹¹⁸³ <https://github.com/NumScale/boost.simd>

¹¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2333>

¹¹⁸⁵ <http://stellar-group.org/2016/09/vectorized-cpp-parallel-algorithms-with-hpx/>

- Added versions of `hpx::get_ptr` taking client side representations for component instances as their parameter (instead of a global id).
- Added the helper utility `hpx::performance_counters::performance_counter_set` helping to encapsulate a set of performance counters to be managed concurrently.
- All execution policies and related classes have been renamed to be consistent with the naming changes applied for C++17. All policies now live in the namespace `hpx::parallel::execution`. The old names can be still enabled at configure time by specifying `-DHPX_WITH_EXECUTION_POLICY_COMPATIBILITY=On` to CMake.
- The thread scheduling subsystem has undergone a major refactoring which results in significant performance improvements. We have also improved the performance of creating `hpx::future` and of various facilities handling those.
- We have consolidated all of the code in `HPX.Compute` related to the integration of CUDA. `hpx::partitioned_vector` has been enabled to be usable with `hpx::compute::vector` which allows 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](#)¹¹⁸⁶ - Adding apex data
- [PR #2595](#)¹¹⁸⁷ - Remove obsolete file
- [Issue #2594](#)¹¹⁸⁸ - FindOpenCL.cmake mismatch with the official cmake module
- [PR #2592](#)¹¹⁸⁹ - First attempt to introduce `spmd_block` in `hpx`

¹¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2596>

¹¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2595>

¹¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2594>

¹¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2592>

- [Issue #2591](#)¹¹⁹⁰ - Feature request: continuation (then) which does not require the callable object to take a `future<R>` as parameter
- [PR #2588](#)¹¹⁹¹ - Daint fixes
- [PR #2587](#)¹¹⁹² - Fixing `transfer_(continuation)_action::schedule`
- [PR #2585](#)¹¹⁹³ - Work around MSVC having an ICE when compiling with `-Ob2`
- [PR #2583](#)¹¹⁹⁴ - chaning 7zip command to 7za in `roll_release.sh`
- [PR #2582](#)¹¹⁹⁵ - First attempt to introduce `spmd_block` in `hpx`
- [PR #2581](#)¹¹⁹⁶ - Enable annotated function for parallel algorithms
- [PR #2580](#)¹¹⁹⁷ - First attempt to introduce `spmd_block` in `hpx`
- [PR #2579](#)¹¹⁹⁸ - Make thread NICE level setting an option
- [PR #2578](#)¹¹⁹⁹ - Implementing enqueue instead of busy wait when no sender is available
- [PR #2577](#)¹²⁰⁰ - Retrieve `-std=c++11` consistent `nvcc` flag
- [PR #2576](#)¹²⁰¹ - Add missing dependencies of cuda based tests
- [PR #2575](#)¹²⁰² - Remove warnings due to some captured variables
- [PR #2573](#)¹²⁰³ - Attempt to resolve `resolve_locality`
- [PR #2572](#)¹²⁰⁴ - Adding APEX hooks to background thread
- [PR #2571](#)¹²⁰⁵ - Pick up `hpx.ignore_batch_env` from config map
- [PR #2570](#)¹²⁰⁶ - Add cmdline options `-hpx:print-counters-locally`
- [PR #2569](#)¹²⁰⁷ - Fix `computeapi` unit tests
- [PR #2567](#)¹²⁰⁸ - This adds another `barrier::synchronize` before registering performance counters
- [PR #2564](#)¹²⁰⁹ - Cray static toolchain support
- [PR #2563](#)¹²¹⁰ - Fixed unhandled exception during startup
- [PR #2562](#)¹²¹¹ - Remove `partitioned_vector.cu` from build tree when `nvcc` is used
- [Issue #2561](#)¹²¹² - octo-tiger crash with commit `6e921495ff6c26f125d62629cbaad0525f14f7ab`

¹¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2591>

¹¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2588>

¹¹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2587>

¹¹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2585>

¹¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2583>

¹¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2582>

¹¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2581>

¹¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2580>

¹¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2579>

¹¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2578>

¹²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2577>

¹²⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2576>

¹²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2575>

¹²⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2573>

¹²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2572>

¹²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2571>

¹²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2570>

¹²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2569>

¹²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2567>

¹²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2564>

¹²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2563>

¹²¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2562>

¹²¹² <https://github.com/STELLAR-GROUP/hpx/issues/2561>

- PR #2560¹²¹³ - Prevent -Wundef warnings on Vc version checks
- PR #2559¹²¹⁴ - Allowing CUDA callback to set the future directly from an OS thread
- PR #2558¹²¹⁵ - Remove warnings due to float precisions
- PR #2557¹²¹⁶ - Removing bogus handling of compile flags for CUDA
- PR #2556¹²¹⁷ - Fixing scan partitioner
- PR #2554¹²¹⁸ - Add more diagnostics to error thrown from find_appropriate_destination
- Issue #2555¹²¹⁹ - No valid parcellport configured
- PR #2553¹²²⁰ - Add cmake cuda_arch option
- PR #2552¹²²¹ - Remove incomplete datapar bindings to libflatarray
- PR #2551¹²²² - Rename hwloc_topology to hwloc_topology_info
- PR #2550¹²²³ - Apex api updates
- PR #2549¹²²⁴ - Pre-include defines.hpp to get the macro HPX_HAVE_CUDA value
- PR #2548¹²²⁵ - Fixing issue with disconnect
- PR #2546¹²²⁶ - Some fixes around cuda clang partitioned_vector example
- PR #2545¹²²⁷ - Fix uses of the Vc2 datapar flags; the value, not the type, should be passed to functions
- PR #2542¹²²⁸ - Make HPX_WITH_MALLOC easier to use
- PR #2541¹²²⁹ - avoid recompiles when enabling/disabling examples
- PR #2540¹²³⁰ - Fixing usage of target_link_libraries()
- PR #2539¹²³¹ - fix RPATH behaviour
- Issue #2538¹²³² - HPX_WITH_CUDA corrupts compilation flags
- PR #2537¹²³³ - Add output of a Bazel Skylark extension for paths and compile options
- PR #2536¹²³⁴ - Add counter exposing total available memory to Windows as well
- PR #2535¹²³⁵ - Remove obsolete support for security

¹²¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2560>

¹²¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2559>

¹²¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2558>

¹²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2557>

¹²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2556>

¹²¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2554>

¹²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2555>

¹²²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2553>

¹²²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2552>

¹²²² <https://github.com/STELLAR-GROUP/hpx/pull/2551>

¹²²³ <https://github.com/STELLAR-GROUP/hpx/pull/2550>

¹²²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2549>

¹²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2548>

¹²²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2546>

¹²²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2545>

¹²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2542>

¹²²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2541>

¹²³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2540>

¹²³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2539>

¹²³² <https://github.com/STELLAR-GROUP/hpx/issues/2538>

¹²³³ <https://github.com/STELLAR-GROUP/hpx/pull/2537>

¹²³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2536>

¹²³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2535>

- Issue #2534¹²³⁶ - Remove command line option `--hpx:run-agas-server`
- PR #2533¹²³⁷ - Pre-cache locality endpoints during bootstrap
- PR #2532¹²³⁸ - Fixing handling of GIDs during serialization preprocessing
- PR #2531¹²³⁹ - Amend uses of the term “functor”
- PR #2529¹²⁴⁰ - added counter for reading available memory
- PR #2527¹²⁴¹ - Facilities to create actions from lambdas
- PR #2526¹²⁴² - Updated docs: HPX_WITH_EXAMPLES
- PR #2525¹²⁴³ - Remove warnings related to unused captured variables
- Issue #2524¹²⁴⁴ - CMAKE failed because it is missing: `TCMALLOC_LIBRARY` `TCMAL-LOC_INCLUDE_DIR`
- PR #2523¹²⁴⁵ - Fixing `compose_cb` stack overflow
- PR #2522¹²⁴⁶ - Instead of unlocking, ignore the lock while creating the message handler
- PR #2521¹²⁴⁷ - Create `LPROGRESS_` logging macro to simplify progress tracking and timings
- PR #2520¹²⁴⁸ - Intel 17 support
- PR #2519¹²⁴⁹ - Fix components example
- PR #2518¹²⁵⁰ - Fixing parcel scheduling
- Issue #2517¹²⁵¹ - Race condition during Parcel Coalescing Handler creation
- Issue #2516¹²⁵² - HPX locks up when using at least 256 localities
- Issue #2515¹²⁵³ - error: Install cannot find “/lib/hpx/libparcel_coalescing.so.0.9.99” but I can see that file
- PR #2514¹²⁵⁴ - Making sure that all continuations of a shared_future are invoked in order
- PR #2513¹²⁵⁵ - Fixing locks held during suspension
- PR #2512¹²⁵⁶ - MPI Parcelport improvements and fixes related to the background work changes
- PR #2511¹²⁵⁷ - Fixing bit-wise (zero-copy) serialization
- Issue #2509¹²⁵⁸ - Linking errors in `hwloc_topology`

¹²³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2534>

¹²³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2533>

¹²³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2532>

¹²³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2531>

¹²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2529>

¹²⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2527>

¹²⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2526>

¹²⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2525>

¹²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2524>

¹²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2523>

¹²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2522>

¹²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2521>

¹²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2520>

¹²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2519>

¹²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2518>

¹²⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2517>

¹²⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2516>

¹²⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/2515>

¹²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2514>

¹²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2513>

¹²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2512>

¹²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2511>

¹²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2509>

- PR #2508¹²⁵⁹ - Added documentation for debugging with core files
- PR #2506¹²⁶⁰ - Fixing background work invocations
- PR #2505¹²⁶¹ - Fix tuple serialization
- Issue #2504¹²⁶² - Ensure continuations are called in the order they have been attached
- PR #2503¹²⁶³ - Adding serialization support for Vc v2 (datapar)
- PR #2502¹²⁶⁴ - Resolve various, minor compiler warnings
- PR #2501¹²⁶⁵ - Some other fixes around cuda examples
- Issue #2500¹²⁶⁶ - nvcc / cuda clang issue due to a missing -DHPX_WITH_CUDA flag
- PR #2499¹²⁶⁷ - Adding support for std::array to wait_all and friends
- PR #2498¹²⁶⁸ - Execute background work as HPX thread
- PR #2497¹²⁶⁹ - Fixing configuration options for spinlock-deadlock detection
- PR #2496¹²⁷⁰ - Accounting for different compilers in CrayKNL toolchain file
- PR #2494¹²⁷¹ - Adding component base class which ties a component instance to a given executor
- PR #2493¹²⁷² - Enable controlling amount of pending threads which must be available to allow thread stealing
- PR #2492¹²⁷³ - Adding new command line option -hpx:print-counter-reset
- PR #2491¹²⁷⁴ - Resolve ambiguities when compiling with APEX
- PR #2490¹²⁷⁵ - Resuming threads waiting on future with higher priority
- Issue #2489¹²⁷⁶ - nvcc issue because -std=c++11 appears twice
- PR #2488¹²⁷⁷ - Adding performance counters exposing the internal idle and busy-loop counters
- PR #2487¹²⁷⁸ - Allowing for plain suspend to reschedule thread right away
- PR #2486¹²⁷⁹ - Only flag HPX code for CUDA if HPX_WITH_CUDA is set
- PR #2485¹²⁸⁰ - Making thread-queue parameters runtime-configurable
- PR #2484¹²⁸¹ - Added atomic counter for parcel-destinations

¹²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2508>

¹²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2506>

¹²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2505>

¹²⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2504>

¹²⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2503>

¹²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2502>

¹²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2501>

¹²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2500>

¹²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2499>

¹²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2498>

¹²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2497>

¹²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2496>

¹²⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2494>

¹²⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2493>

¹²⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2492>

¹²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2491>

¹²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2490>

¹²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2489>

¹²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2488>

¹²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2487>

¹²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2486>

¹²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2485>

¹²⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2484>

- PR #2483¹²⁸² - Added priority-queue lifo scheduler
- PR #2482¹²⁸³ - Changing scheduler to steal only if more than a minimal number of tasks are available
- PR #2481¹²⁸⁴ - Extending command line option `-hpx:print-counter-destination` to support value 'none'
- PR #2479¹²⁸⁵ - Added option to disable signal handler
- PR #2478¹²⁸⁶ - Making sure the sine performance counter module gets loaded only for the corresponding example
- Issue #2477¹²⁸⁷ - Breaking at a throw statement
- PR #2476¹²⁸⁸ - Annotated function
- PR #2475¹²⁸⁹ - Ensure that using `%osthread%` during logging will not throw for non-hpx threads
- PR #2474¹²⁹⁰ - Remove now superficial `non_direct` actions from `base_lco` and friends
- PR #2473¹²⁹¹ - Refining support for `ITTNotify`
- PR #2472¹²⁹² - Some fixes around `hpx compute`
- Issue #2470¹²⁹³ - redefinition of `boost::detail::spinlock`
- Issue #2469¹²⁹⁴ - Dataflow performance issue
- PR #2468¹²⁹⁵ - Perf docs update
- PR #2466¹²⁹⁶ - Guarantee to execute remote direct actions on HPX-thread
- PR #2465¹²⁹⁷ - Improve demo : Async copy and fixed device handling
- PR #2464¹²⁹⁸ - Adding performance counter exposing instantaneous scheduler utilization
- PR #2463¹²⁹⁹ - Downcast to `future<void>`
- PR #2462¹³⁰⁰ - Fixed usage of `ITT-Notify` API with Intel Amplifier
- PR #2461¹³⁰¹ - Cublas demo
- PR #2460¹³⁰² - Fixing thread bindings
- PR #2459¹³⁰³ - Make `-std=c++11` nvcc flag consistent for in-build and installed versions
- Issue #2457¹³⁰⁴ - Segmentation fault when registering a partitioned vector

¹²⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2483>

¹²⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2482>

¹²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2481>

¹²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2479>

¹²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2478>

¹²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2477>

¹²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2476>

¹²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2475>

¹²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2474>

¹²⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2473>

¹²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2472>

¹²⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/2470>

¹²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2469>

¹²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2468>

¹²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2466>

¹²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2465>

¹²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2464>

¹²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2463>

¹³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2462>

¹³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2461>

¹³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2460>

¹³⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2459>

¹³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2457>

- PR #2452¹³⁰⁵ - Properly releasing global barrier for unhandled exceptions
- PR #2451¹³⁰⁶ - Fixing long shutdown times
- PR #2450¹³⁰⁷ - Attempting to fix initialization errors on newer platforms (Boost V1.63)
- PR #2449¹³⁰⁸ - Replace BOOST_COMPILER_FENCE with an HPX version
- PR #2448¹³⁰⁹ - This fixes a possible race in the migration code
- **PR #2445¹³¹⁰ - Fixing dataflow et.al. for futures or future-ranges wrapped into ref()**
- PR #2444¹³¹¹ - Fix segfaults
- PR #2443¹³¹² - Issue 2442
- Issue #2442¹³¹³ - Mismatch between #if/#endif and namespace scope brackets in this_thread_executors.hpp
- Issue #2441¹³¹⁴ - undeclared identifier BOOST_COMPILER_FENCE
- PR #2440¹³¹⁵ - Knl build
- PR #2438¹³¹⁶ - Datapar backend
- PR #2437¹³¹⁷ - Adapt algorithm parameter sequence changes from C++17
- PR #2436¹³¹⁸ - Adapt execution policy name changes from C++17
- Issue #2435¹³¹⁹ - Trunk broken, undefined reference to hpx::thread::interrupt(hpx::thread::id, bool)
- PR #2434¹³²⁰ - More fixes to resource manager
- PR #2433¹³²¹ - Added versions of hpx::get_ptr taking client side representations
- PR #2432¹³²² - Warning fixes
- PR #2431¹³²³ - Adding facility representing set of performance counters
- PR #2430¹³²⁴ - Fix parallel_executor thread spawning
- PR #2429¹³²⁵ - Fix attribute warning for gcc
- Issue #2427¹³²⁶ - Seg fault running octo-tiger with latest HPX commit
- Issue #2426¹³²⁷ - Bug in 9592f5c0bc29806fce0dbe73f35b6ca7e027edcb causes immediate crash in Octo-tiger

¹³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2452>

¹³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2451>

¹³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2450>

¹³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2449>

¹³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2448>

¹³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2445>

¹³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2444>

¹³¹² <https://github.com/STELLAR-GROUP/hpx/pull/2443>

¹³¹³ <https://github.com/STELLAR-GROUP/hpx/issues/2442>

¹³¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2441>

¹³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2440>

¹³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2438>

¹³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2437>

¹³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2436>

¹³¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2435>

¹³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2434>

¹³²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2433>

¹³²² <https://github.com/STELLAR-GROUP/hpx/pull/2432>

¹³²³ <https://github.com/STELLAR-GROUP/hpx/pull/2431>

¹³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2430>

¹³²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2429>

¹³²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2427>

¹³²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2426>

- PR #2425¹³²⁸ - Fix nvcc errors due to constexpr specifier
- Issue #2424¹³²⁹ - Async action on component present on `hpx::find_here` is executing synchronously
- PR #2423¹³³⁰ - Fix nvcc errors due to constexpr specifier
- PR #2422¹³³¹ - Implementing `hpx::this_thread` thread data functions
- PR #2421¹³³² - Adding benchmark for `wait_all`
- Issue #2420¹³³³ - Returning object of a component client from another component action fails
- PR #2419¹³³⁴ - Infiniband parselport
- Issue #2418¹³³⁵ - gcc + nvcc fails to compile code that uses `partitioned_vector`
- PR #2417¹³³⁶ - Fixing context switching
- PR #2416¹³³⁷ - Adding fixes and workarounds to allow compilation with nvcc/msvc (VS2015up3)
- PR #2415¹³³⁸ - Fix errors coming from hpx compute examples
- PR #2414¹³³⁹ - Fixing msvc12
- PR #2413¹³⁴⁰ - Enable cuda/nvcc or cuda/clang when using `add_hpx_executable()`
- PR #2412¹³⁴¹ - Fix issue in `HPX_SetupTarget.cmake` when cuda is used
- PR #2411¹³⁴² - This fixes the core compilation issues with MSVC12
- Issue #2410¹³⁴³ - undefined reference to `opal_hwloc191_hwloc_.....`
- PR #2409¹³⁴⁴ - Fixing locking for channel and `receive_buffer`
- PR #2407¹³⁴⁵ - Solving #2402 and #2403
- PR #2406¹³⁴⁶ - Improve guards
- PR #2405¹³⁴⁷ - Enable `parallel::for_each` for iterators returning proxy types
- PR #2404¹³⁴⁸ - Forward the explicitly given `result_type` in the hpx invoke
- Issue #2403¹³⁴⁹ - `datapar_execution` + zip iterator: lambda arguments aren't references
- Issue #2402¹³⁵⁰ - `datapar` algorithm instantiated with wrong type #2402

¹³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2425>

¹³²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2424>

¹³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2423>

¹³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2422>

¹³³² <https://github.com/STELLAR-GROUP/hpx/pull/2421>

¹³³³ <https://github.com/STELLAR-GROUP/hpx/issues/2420>

¹³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2419>

¹³³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2418>

¹³³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2417>

¹³³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2416>

¹³³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2415>

¹³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2414>

¹³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2413>

¹³⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2412>

¹³⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2411>

¹³⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2410>

¹³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2409>

¹³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2407>

¹³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2406>

¹³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2405>

¹³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2404>

¹³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2403>

¹³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2402>

- PR #2401¹³⁵¹ - Added support for imported libraries to HPX_Libraries.cmake
- PR #2400¹³⁵² - Use CMake policy CMP0060
- Issue #2399¹³⁵³ - Error trying to push back vector of futures to vector
- PR #2398¹³⁵⁴ - Allow config #defines to be written out to custom config/defines.hpp
- Issue #2397¹³⁵⁵ - CMake generated config defines can cause tedious rebuilds category
- Issue #2396¹³⁵⁶ - BOOST_ROOT paths are not used at link time
- PR #2395¹³⁵⁷ - Fix target_link_libraries() issue when HPX Cuda is enabled
- Issue #2394¹³⁵⁸ - Template compilation error using HPX_WITH_DATAPAR_LIBFLATARRAY
- PR #2393¹³⁵⁹ - Fixing lock registration for recursive mutex
- PR #2392¹³⁶⁰ - Add keywords in target_link_libraries in hpx_setup_target
- PR #2391¹³⁶¹ - Clang goroutines
- Issue #2390¹³⁶² - Adapt execution policy name changes from C++17
- PR #2389¹³⁶³ - Chunk allocator and pool are not used and are obsolete
- PR #2388¹³⁶⁴ - Adding functionalities to datapar needed by octotiger
- PR #2387¹³⁶⁵ - Fixing race condition for early parcels
- Issue #2386¹³⁶⁶ - Lock registration broken for recursive_mutex
- PR #2385¹³⁶⁷ - Datapar zip iterator
- PR #2384¹³⁶⁸ - Fixing race condition in for_loop_reduction
- PR #2383¹³⁶⁹ - Continuations
- PR #2382¹³⁷⁰ - add LibFlatArray-based backend for datapar
- PR #2381¹³⁷¹ - remove unused typedef to get rid of compiler warnings
- PR #2380¹³⁷² - Tau cleanup
- PR #2379¹³⁷³ - Can send immediate

¹³⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2401>
¹³⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2400>
¹³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/2399>
¹³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2398>
¹³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2397>
¹³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2396>
¹³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2395>
¹³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2394>
¹³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2393>
¹³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2392>
¹³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2391>
¹³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2390>
¹³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2389>
¹³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2388>
¹³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2387>
¹³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2386>
¹³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2385>
¹³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2384>
¹³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2383>
¹³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2382>
¹³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2381>
¹³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2380>
¹³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2379>

- PR #2378¹³⁷⁴ - Renaming copy_helper/copy_n_helper/move_helper/move_n_helper
- Issue #2376¹³⁷⁵ - Boost trunk's spinlock initializer fails to compile
- PR #2375¹³⁷⁶ - Add support for minimal thread local data
- PR #2374¹³⁷⁷ - Adding API functions set_config_entry_callback
- PR #2373¹³⁷⁸ - Add a simple utility for debugging that gives suspended task backtraces
- PR #2372¹³⁷⁹ - Barrier Fixes
- Issue #2370¹³⁸⁰ - Can't wait on a wrapped future
- PR #2369¹³⁸¹ - Fixing stable_partition
- PR #2367¹³⁸² - Fixing find_prefixes for Windows platforms
- PR #2366¹³⁸³ - Testing for experimental/optional only in C++14 mode
- PR #2364¹³⁸⁴ - Adding set_config_entry
- PR #2363¹³⁸⁵ - Fix papi
- PR #2362¹³⁸⁶ - Adding missing macros for new non-direct actions
- PR #2361¹³⁸⁷ - Improve cmake output to help debug compiler incompatibility check
- PR #2360¹³⁸⁸ - Fixing race condition in condition_variable
- PR #2359¹³⁸⁹ - Fixing shutdown when parcels are still in flight
- Issue #2357¹³⁹⁰ - failed to insert console_print_action into typename_to_id_t registry
- PR #2356¹³⁹¹ - Fixing return type of get_iterator_tuple
- PR #2355¹³⁹² - Fixing compilation against Boost 1.62
- PR #2354¹³⁹³ - Adding serialization for mask_type if CPU_COUNT > 64
- PR #2353¹³⁹⁴ - Adding hooks to tie in APEX into the parcel layer
- Issue #2352¹³⁹⁵ - Compile errors when using intel 17 beta (for KNL) on edison
- PR #2351¹³⁹⁶ - Fix function vtable get_function_address implementation

¹³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2378>

¹³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2376>

¹³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2375>

¹³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2374>

¹³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2373>

¹³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2372>

¹³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2370>

¹³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2369>

¹³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2367>

¹³⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2366>

¹³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2364>

¹³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2363>

¹³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2362>

¹³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2361>

¹³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2360>

¹³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2359>

¹³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2357>

¹³⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2356>

¹³⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2355>

¹³⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2354>

¹³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2353>

¹³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2352>

¹³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2351>

- Issue #2350¹³⁹⁷ - Build failure - master branch (4de09f5) with Intel Compiler v17
- PR #2349¹³⁹⁸ - Enabling zero-copy serialization support for `std::vector<>`
- PR #2348¹³⁹⁹ - Adding test to verify #2334 is fixed
- PR #2347¹⁴⁰⁰ - Bug fixes for `hpx.compute` and `hpx::lcos::channel`
- PR #2346¹⁴⁰¹ - Removing cmake “find” files that are in the APEX cmake Modules
- PR #2345¹⁴⁰² - Implemented `parallel::stable_partition`
- PR #2344¹⁴⁰³ - Making `hpx::lcos::channel` usable with basename registration
- PR #2343¹⁴⁰⁴ - Fix a couple of examples that failed to compile after recent api changes
- Issue #2342¹⁴⁰⁵ - Enabling APEX causes link errors
- PR #2341¹⁴⁰⁶ - Removing cmake “find” files that are in the APEX cmake Modules
- PR #2340¹⁴⁰⁷ - Implemented all existing datapar algorithms using Boost.SIMD
- PR #2339¹⁴⁰⁸ - Fixing 2338
- PR #2338¹⁴⁰⁹ - Possible race in sliding semaphore
- PR #2337¹⁴¹⁰ - Adjust `osu_latency` test to measure `window_size` parcels in flight at once
- PR #2336¹⁴¹¹ - Allowing remote direct actions to be executed without spawning a task
- PR #2335¹⁴¹² - Making sure multiple components are properly initialized from arguments
- Issue #2334¹⁴¹³ - Cannot construct component with large vector on a remote locality
- PR #2332¹⁴¹⁴ - Fixing `hpx::lcos::local::barrier`
- PR #2331¹⁴¹⁵ - Updating APEX support to include OTF2
- PR #2330¹⁴¹⁶ - Support for data-parallelism for parallel algorithms
- Issue #2329¹⁴¹⁷ - Coordinate settings in cmake
- PR #2328¹⁴¹⁸ - fix LibGeoDecomp builds with HPX + GCC 5.3.0 + CUDA 8RC
- PR #2326¹⁴¹⁹ - Making `scan_partitioner` work (for now)

¹³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2350>

¹³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2349>

¹³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2348>

¹⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2347>

¹⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2346>

¹⁴⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2345>

¹⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2344>

¹⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2343>

¹⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2342>

¹⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2341>

¹⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2340>

¹⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2339>

¹⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2338>

¹⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2337>

¹⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2336>

¹⁴¹² <https://github.com/STELLAR-GROUP/hpx/pull/2335>

¹⁴¹³ <https://github.com/STELLAR-GROUP/hpx/issues/2334>

¹⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2332>

¹⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2331>

¹⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2330>

¹⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2329>

¹⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2328>

¹⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2326>

- [Issue #2323](#)¹⁴²⁰ - Constructing a vector of components only correctly initializes the first component
- [PR #2322](#)¹⁴²¹ - Fix problems that bubbled up after merging #2278
- [PR #2321](#)¹⁴²² - Scalable barrier
- [PR #2320](#)¹⁴²³ - Std flag fixes
- [Issue #2319](#)¹⁴²⁴ - `-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](#)¹⁴²⁵ - Improve handling of `-hpx:bind=<bind-spec>`
- [PR #2317](#)¹⁴²⁶ - Making sure command line warnings are printed once only
- [PR #2316](#)¹⁴²⁷ - Fixing command line handling for default bind mode
- [PR #2315](#)¹⁴²⁸ - Set `id_retrieved` if `set_id` is present
- [Issue #2314](#)¹⁴²⁹ - Warning for requested/allocated thread discrepancy is printed twice
- [Issue #2313](#)¹⁴³⁰ - `-hpx:print-bind` doesn't work with `-hpx:pu-step`
- [Issue #2312](#)¹⁴³¹ - `-hpx:bind` range specifier restrictions are overly restrictive
- [Issue #2311](#)¹⁴³² - `hpx_0.9.99` out of project build fails
- [PR #2310](#)¹⁴³³ - Simplify function registration
- [PR #2309](#)¹⁴³⁴ - Spelling and grammar revisions in documentation (and some code)
- [PR #2306](#)¹⁴³⁵ - Correct minor typo in the documentation
- [PR #2305](#)¹⁴³⁶ - Cleaning up and fixing parcel coalescing
- [PR #2304](#)¹⁴³⁷ - Inspect checks for stream related includes
- [PR #2303](#)¹⁴³⁸ - Add functionality allowing to enumerate threads of given state
- [PR #2301](#)¹⁴³⁹ - Algorithm overloads fix for VS2013
- [PR #2300](#)¹⁴⁴⁰ - Use `<stdint>`, add inspect checks
- [PR #2299](#)¹⁴⁴¹ - Replace `boost::[c]ref` with `std::[c]ref`, add inspect checks
- [PR #2297](#)¹⁴⁴² - Fixing compilation with no `hw_loc`

¹⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2323>

¹⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2322>

¹⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/2321>

¹⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/2320>

¹⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2319>

¹⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2318>

¹⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2317>

¹⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2316>

¹⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2315>

¹⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2314>

¹⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2313>

¹⁴³¹ <https://github.com/STELLAR-GROUP/hpx/issues/2312>

¹⁴³² <https://github.com/STELLAR-GROUP/hpx/issues/2311>

¹⁴³³ <https://github.com/STELLAR-GROUP/hpx/pull/2310>

¹⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2309>

¹⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2306>

¹⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2305>

¹⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2304>

¹⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2303>

¹⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2301>

¹⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2300>

¹⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2299>

¹⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2297>

- PR #2296¹⁴⁴³ - Hpx compute
- PR #2295¹⁴⁴⁴ - Making sure for_loop(execution::par, 0, N, ...) is actually executed in parallel
- PR #2294¹⁴⁴⁵ - Throwing exceptions if the runtime is not up and running
- PR #2293¹⁴⁴⁶ - Removing unused parcel port code
- PR #2292¹⁴⁴⁷ - Refactor function vtables
- PR #2291¹⁴⁴⁸ - Fixing 2286
- PR #2290¹⁴⁴⁹ - Simplify algorithm overloads
- PR #2289¹⁴⁵⁰ - Adding performance counters reporting parcel related data on a per-action basis
- Issue #2288¹⁴⁵¹ - Remove dormant parcelports
- Issue #2286¹⁴⁵² - adjustments to parcel handling to support parcelports that do not need a connection cache
- PR #2285¹⁴⁵³ - add CMake option to disable package export
- PR #2283¹⁴⁵⁴ - Add more inspect checks for use of deprecated components
- Issue #2282¹⁴⁵⁵ - Arithmetic exception in executor static chunker
- Issue #2281¹⁴⁵⁶ - For loop doesn't parallelize
- PR #2280¹⁴⁵⁷ - Fixing 2277: build failure with PAPI
- PR #2279¹⁴⁵⁸ - Child vs parent stealing
- Issue #2277¹⁴⁵⁹ - master branch build failure (53c5b4f) with papi
- PR #2276¹⁴⁶⁰ - Compile time launch policies
- PR #2275¹⁴⁶¹ - Replace boost::chrono with std::chrono in interfaces
- PR #2274¹⁴⁶² - Replace most uses of Boost.Assign with initializer list
- PR #2273¹⁴⁶³ - Fixed typos
- PR #2272¹⁴⁶⁴ - Inspect checks
- PR #2270¹⁴⁶⁵ - Adding test verifying -lhpx.os_threads=all

¹⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2296>

¹⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2295>

¹⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2294>

¹⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2293>

¹⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2292>

¹⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2291>

¹⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2290>

¹⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2289>

¹⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2288>

¹⁴⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2286>

¹⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2285>

¹⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2283>

¹⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2282>

¹⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2281>

¹⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2280>

¹⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2279>

¹⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2277>

¹⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2276>

¹⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2275>

¹⁴⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2274>

¹⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2273>

¹⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2272>

¹⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2270>

- PR #2269¹⁴⁶⁶ - Added inspect check for now obsolete boost type traits
- PR #2268¹⁴⁶⁷ - Moving more code into source files
- Issue #2267¹⁴⁶⁸ - Add inspect support to deprecate Boost.TypeTraits
- PR #2265¹⁴⁶⁹ - Adding channel LCO
- PR #2264¹⁴⁷⁰ - Make support for std::ref mandatory
- PR #2263¹⁴⁷¹ - Constrain tuple_member forwarding constructor
- Issue #2262¹⁴⁷² - Test hpx.os_threads=all
- Issue #2261¹⁴⁷³ - OS X: Error: no matching constructor for initialization of 'hpx::lcos::local::condition_variable_any'
- Issue #2260¹⁴⁷⁴ - Make support for std::ref mandatory
- PR #2259¹⁴⁷⁵ - Remove most of Boost.MPL, Boost.EnableIf and Boost.TypeTraits
- PR #2258¹⁴⁷⁶ - Fixing #2256
- PR #2257¹⁴⁷⁷ - Fixing launch process
- Issue #2256¹⁴⁷⁸ - Actions are not registered if not invoked
- PR #2255¹⁴⁷⁹ - Coalescing histogram
- PR #2254¹⁴⁸⁰ - Silence explicit initialization in copy-constructor warnings
- PR #2253¹⁴⁸¹ - Drop support for GCC 4.6 and 4.7
- PR #2252¹⁴⁸² - Prepare V1.0
- PR #2251¹⁴⁸³ - Convert to 0.9.99
- PR #2249¹⁴⁸⁴ - Adding iterator_facade and iterator_adaptor
- Issue #2248¹⁴⁸⁵ - Need a feature to yield to a new task immediately
- PR #2246¹⁴⁸⁶ - Adding split_future
- PR #2245¹⁴⁸⁷ - Add an example for handing over a component instance to a dynamically launched locality
- Issue #2243¹⁴⁸⁸ - Add example demonstrating AGAS symbolic name registration

¹⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2269>

¹⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2268>

¹⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2267>

¹⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2265>

¹⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2264>

¹⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2263>

¹⁴⁷² <https://github.com/STELLAR-GROUP/hpx/issues/2262>

¹⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/2261>

¹⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2260>

¹⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2259>

¹⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2258>

¹⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2257>

¹⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2256>

¹⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2255>

¹⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2254>

¹⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2253>

¹⁴⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2252>

¹⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2251>

¹⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2249>

¹⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2248>

¹⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2246>

¹⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2245>

¹⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2243>

- Issue #2242¹⁴⁸⁹ - pkgconfig test broken on CentOS 7 / Boost 1.61
- Issue #2241¹⁴⁹⁰ - Compilation error for partitioned vector in hpx_compute branch
- PR #2240¹⁴⁹¹ - Fixing termination detection on one locality
- Issue #2239¹⁴⁹² - Create a new facility lcos::split_all
- Issue #2236¹⁴⁹³ - hpx::cout vs. std::cout
- PR #2232¹⁴⁹⁴ - Implement local-only primary namespace service
- Issue #2147¹⁴⁹⁵ - would like to know how much data is being routed by particular actions
- Issue #2109¹⁴⁹⁶ - Warning while compiling hpx
- Issue #1973¹⁴⁹⁷ - Setting INTERFACE_COMPILE_OPTIONS for hpx_init in CMake taints Fortran_FLAGS
- Issue #1864¹⁴⁹⁸ - run_guarded using bound function ignores reference
- Issue #1754¹⁴⁹⁹ - Running with TCP parselport causes immediate crash or freeze
- Issue #1655¹⁵⁰⁰ - Enable zip_iterator to be used with Boost traversal iterator categories
- Issue #1591¹⁵⁰¹ - Optimize AGAS for shared memory only operation
- Issue #1401¹⁵⁰² - Need an efficient infiniband parselport
- Issue #1125¹⁵⁰³ - Fix the IPC parselport
- Issue #839¹⁵⁰⁴ - Refactor ibverbs and shmем parselport
- Issue #702¹⁵⁰⁵ - Add instrumentation of parcel layer
- Issue #668¹⁵⁰⁶ - Implement ispc task interface
- Issue #533¹⁵⁰⁷ - Thread queue/deque internal parameters should be runtime configurable
- Issue #475¹⁵⁰⁸ - Create a means of combining performance counters into querysets

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

- ¹⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2242>
- ¹⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2241>
- ¹⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2240>
- ¹⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/2239>
- ¹⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/2236>
- ¹⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2232>
- ¹⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2147>
- ¹⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2109>
- ¹⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1973>
- ¹⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1864>
- ¹⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1754>
- ¹⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1655>
- ¹⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1591>
- ¹⁵⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1401>
- ¹⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1125>
- ¹⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/839>
- ¹⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/702>
- ¹⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/668>
- ¹⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/533>
- ¹⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/475>

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 (N4560¹⁵⁰⁹) proposal. This also includes the addition of projections to the existing algorithms. Please see Issue #1668¹⁵¹⁰ for a list of algorithms which have been adapted to N4560¹⁵¹¹.
- We have implemented index-based parallel for-loops based on a corresponding standardization proposal (P0075R1¹⁵¹²). Please see Issue #2016¹⁵¹³ for a list of available algorithms.
- We have added implementations for more parallel algorithms as proposed for the upcoming C++ 17 Standard. See Issue #1141¹⁵¹⁴ 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 #559¹⁵¹⁵ and PR #1966¹⁵¹⁶ for more details.
- The `hpx::dataflow` facility is now usable with actions. Similarly to `hpx::async`, actions can be specified as an explicit template argument (`hpx::dataflow<Action>(target, ...)`) or as the first argument (`hpx::dataflow(Action(), target, ...)`). We have also enabled the use of distribution policies as the target for dataflow invocations. Please see Issue #1265¹⁵¹⁷ and PR #1912¹⁵¹⁸ for more information.

¹⁵⁰⁹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf>

¹⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1668>

¹⁵¹¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf>

¹⁵¹² <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0075r1.pdf>

¹⁵¹³ <https://github.com/STELLAR-GROUP/hpx/issues/2016>

¹⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1141>

¹⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/559>

¹⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1966>

¹⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1265>

¹⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1912>

- 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](#)¹⁵¹⁹) 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`¹⁵²⁰ 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](#)¹⁵²¹) proposal imposes some breaking changes related to the return types of some of the parallel algorithms. Please see [Issue #1668](#)¹⁵²² for a list of algorithms which have already been adapted.
- The facility `hpx::lcos::make_future_void()` has been replaced by `hpx::make_future<void>()`.
- We have removed support for Intel V13 and gcc 4.4.x.
- We have removed (default) support for the generic `hpx::parallel::execution_policy` because it was removed from the Parallelism TS (`__cpp11_n4104__`) while it was being added to the upcoming C++17 Standard. This facility can be still enabled at configure time by specifying `-DHPX_WITH_GENERIC_EXECUTION_POLICY=On` to `CMake`.
- Uses of `boost::shared_ptr` and related facilities have been replaced with `std::shared_ptr` and friends. Uses of `boost::unique_lock`, `boost::lock_guard` etc. have also been replaced by the equivalent (and equally named) tools available from the C++11 standard library.
- Facilities that used to expect an explicit `boost::unique_lock` now take an `std::unique_lock`. Additionally, `condition_variable` no longer aliases `condition_variable_any`; its interface now only works with `std::unique_lock<local::mutex>`.

¹⁵¹⁹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf>

¹⁵²⁰ <https://www.cmake.org>

¹⁵²¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf>

¹⁵²² <https://github.com/STELLAR-GROUP/hpx/issues/1668>

- 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 #2250¹⁵²³ - change default chunker of parallel executor to static one
- PR #2247¹⁵²⁴ - HPX on ppc64le
- PR #2244¹⁵²⁵ - Fixing MSVC problems
- PR #2238¹⁵²⁶ - Fixing small typos
- PR #2237¹⁵²⁷ - Fixing small typos
- PR #2234¹⁵²⁸ - Fix broken add test macro when extra args are passed in
- PR #2231¹⁵²⁹ - Fixing possible race during future awaiting in serialization
- PR #2230¹⁵³⁰ - Fix stream nvcc
- PR #2229¹⁵³¹ - Fixed `run_as_hpx_thread`
- PR #2228¹⁵³² - On `prefetching_test` branch : adding `prefetching_iterator` and related tests used for prefetching containers within lambda functions
- PR #2227¹⁵³³ - Support for HPXCL's `opencl::event`
- PR #2226¹⁵³⁴ - Preparing for release of V0.9.99
- PR #2225¹⁵³⁵ - fix issue when compiling components with `hpxcxx`
- PR #2224¹⁵³⁶ - Compute alloc fix
- PR #2223¹⁵³⁷ - Simplify promise
- PR #2222¹⁵³⁸ - Replace last uses of `boost::function` by `util::function_nonsr`
- PR #2221¹⁵³⁹ - Fix config tests
- PR #2220¹⁵⁴⁰ - Fixing gcc 4.6 compilation issues
- PR #2219¹⁵⁴¹ - `nullptr` support for `[unique_]function`

¹⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/2250>

¹⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2247>

¹⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2244>

¹⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2238>

¹⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2237>

¹⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2234>

¹⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2231>

¹⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2230>

¹⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2229>

¹⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/2228>

¹⁵³³ <https://github.com/STELLAR-GROUP/hpx/pull/2227>

¹⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2226>

¹⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2225>

¹⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2224>

¹⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2223>

¹⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2222>

¹⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2221>

¹⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2220>

¹⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2219>

- PR #2218¹⁵⁴² - Introducing clang tidy
- PR #2216¹⁵⁴³ - Replace NULL with nullptr
- Issue #2214¹⁵⁴⁴ - Let inspect flag use of NULL, suggest nullptr instead
- PR #2213¹⁵⁴⁵ - Require support for nullptr
- PR #2212¹⁵⁴⁶ - Properly find jemalloc through pkg-config
- PR #2211¹⁵⁴⁷ - Disable a couple of warnings reported by Intel on Windows
- PR #2210¹⁵⁴⁸ - Fixed host::block_allocator::bulk_construct
- PR #2209¹⁵⁴⁹ - Started to clean up new sort algorithms, made things compile for sort_by_key
- PR #2208¹⁵⁵⁰ - A couple of fixes that were exposed by a new sort algorithm
- PR #2207¹⁵⁵¹ - Adding missing includes in /hpx/include/serialization.hpp
- PR #2206¹⁵⁵² - Call package_action::get_future before package_action::apply
- PR #2205¹⁵⁵³ - The indirect_packaged_task::operator() needs to be run on a HPX thread
- PR #2204¹⁵⁵⁴ - Variadic executor parameters
- PR #2203¹⁵⁵⁵ - Delay-initialize members of partitioned iterator
- PR #2202¹⁵⁵⁶ - Added segmented fill for hpx::vector
- Issue #2201¹⁵⁵⁷ - Null Thread id encountered on partitioned_vector
- PR #2200¹⁵⁵⁸ - Fix hangs
- PR #2199¹⁵⁵⁹ - Deprecating hpx/traits.hpp
- PR #2198¹⁵⁶⁰ - Making explicit inclusion of external libraries into build
- PR #2197¹⁵⁶¹ - Fix typo in QT CMakeLists
- PR #2196¹⁵⁶² - Fixing a gcc warning about attributes being ignored
- PR #2194¹⁵⁶³ - Fixing partitioned_vector_spmd_foreach example
- Issue #2193¹⁵⁶⁴ - partitioned_vector_spmd_foreach seg faults

¹⁵⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2218>
¹⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2216>
¹⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2214>
¹⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2213>
¹⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2212>
¹⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2211>
¹⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2210>
¹⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2209>
¹⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2208>
¹⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2207>
¹⁵⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2206>
¹⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2205>
¹⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2204>
¹⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2203>
¹⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2202>
¹⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2201>
¹⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2200>
¹⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2199>
¹⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2198>
¹⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2197>
¹⁵⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2196>
¹⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2194>
¹⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2193>

- PR #2192¹⁵⁶⁵ - Support Boost.Thread v4
- PR #2191¹⁵⁶⁶ - HPX.Compute prototype
- PR #2190¹⁵⁶⁷ - Spawning operation on new thread if remaining stack space becomes too small
- PR #2189¹⁵⁶⁸ - Adding callback taking index and future to when_each
- PR #2188¹⁵⁶⁹ - Adding new example demonstrating receive_buffer
- PR #2187¹⁵⁷⁰ - Mask 128-bit ints if CUDA is being used
- PR #2186¹⁵⁷¹ - Make startup & shutdown functions unique_function
- PR #2185¹⁵⁷² - Fixing logging output not to cause hang on shutdown
- PR #2184¹⁵⁷³ - Allowing component clients as action return types
- Issue #2183¹⁵⁷⁴ - Enabling logging output causes hang on shutdown
- Issue #2182¹⁵⁷⁵ - 1d_stencil seg fault
- Issue #2181¹⁵⁷⁶ - Setting small stack size does not change default
- PR #2180¹⁵⁷⁷ - Changing default bind mode to balanced
- PR #2179¹⁵⁷⁸ - adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2177¹⁵⁷⁹ - Fixing 2176
- Issue #2176¹⁵⁸⁰ - Launch process test fails on OSX
- PR #2175¹⁵⁸¹ - Fix unbalanced config/warnings includes, add some new ones
- PR #2174¹⁵⁸² - Fix test categorization : regression not unit
- Issue #2172¹⁵⁸³ - Different performance results
- Issue #2171¹⁵⁸⁴ - “negative entry in reference count table” running octotiger on 32 nodes on queenbee
- Issue #2170¹⁵⁸⁵ - Error while compiling on Mac + boost 1.60
- PR #2168¹⁵⁸⁶ - Fixing problems with is_bitwise_serializable
- Issue #2167¹⁵⁸⁷ - startup & shutdown function should accept unique_function

¹⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2192>

¹⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2191>

¹⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2190>

¹⁵⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2189>

¹⁵⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2188>

¹⁵⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2187>

¹⁵⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2186>

¹⁵⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2185>

¹⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2184>

¹⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2183>

¹⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2182>

¹⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2181>

¹⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2180>

¹⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2179>

¹⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2177>

¹⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2176>

¹⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2175>

¹⁵⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2174>

¹⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/2172>

¹⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2171>

¹⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2170>

¹⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2168>

¹⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2167>

- Issue #2166¹⁵⁸⁸ - Simple receive_buffer example
- PR #2165¹⁵⁸⁹ - Fix wait all
- PR #2164¹⁵⁹⁰ - Fix wait all
- PR #2163¹⁵⁹¹ - Fix some typos in config tests
- PR #2162¹⁵⁹² - Improve #includes
- PR #2160¹⁵⁹³ - Add inspect check for missing #include <list>
- PR #2159¹⁵⁹⁴ - Add missing finalize call to stop test hanging
- PR #2158¹⁵⁹⁵ - Algo fixes
- PR #2157¹⁵⁹⁶ - Stack check
- Issue #2156¹⁵⁹⁷ - OSX reports stack space incorrectly (generic context coroutines)
- Issue #2155¹⁵⁹⁸ - Race condition suspected in runtime
- PR #2154¹⁵⁹⁹ - Replace boost::detail::atomic_count with the new util::atomic_count
- PR #2153¹⁶⁰⁰ - Fix stack overflow on OSX
- PR #2152¹⁶⁰¹ - Define is_bitwise_serializable as is_trivially_copyable when available
- PR #2151¹⁶⁰² - Adding missing <cstring> for std::mem* functions
- Issue #2150¹⁶⁰³ - Unable to use component clients as action return types
- PR #2149¹⁶⁰⁴ - std::memmove copies bytes, use bytes*sizeof(type) when copying larger types
- PR #2146¹⁶⁰⁵ - Adding customization point for parallel copy/move
- PR #2145¹⁶⁰⁶ - Applying changes to address warnings issued by latest version of PVS Studio
- Issue #2148¹⁶⁰⁷ - hpx::parallel::copy is broken after trivially copyable changes
- PR #2144¹⁶⁰⁸ - Some minor tweaks to compute prototype
- PR #2143¹⁶⁰⁹ - Added Boost version support information over OSX platform
- PR #2142¹⁶¹⁰ - Fixing memory leak in example

¹⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2166>

¹⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2165>

¹⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2164>

¹⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2163>

¹⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2162>

¹⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2160>

¹⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2159>

¹⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2158>

¹⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2157>

¹⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2156>

¹⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2155>

¹⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2154>

¹⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2153>

¹⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2152>

¹⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2151>

¹⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2150>

¹⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2149>

¹⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2146>

¹⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2145>

¹⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2148>

¹⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2144>

¹⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2143>

¹⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2142>

- PR #2141¹⁶¹¹ - Add missing specializations in execution policies
- PR #2139¹⁶¹² - This PR fixes a few problems reported by Clang’s Undefined Behavior sanitizer
- PR #2138¹⁶¹³ - Revert “Adding fedora docs”
- PR #2136¹⁶¹⁴ - Removed double semicolon
- PR #2135¹⁶¹⁵ - Add deprecated #include check for hpx_fwd.hpp
- PR #2134¹⁶¹⁶ - Resolved memory leak in stencil_8
- PR #2133¹⁶¹⁷ - Replace uses of boost pointer containers
- PR #2132¹⁶¹⁸ - Removing unused typedef
- PR #2131¹⁶¹⁹ - Add several include checks for std facilities
- PR #2130¹⁶²⁰ - Fixing parcel compression, adding test
- PR #2129¹⁶²¹ - Fix invalid attribute warnings
- Issue #2128¹⁶²² - hpx::init seems to segfault
- PR #2127¹⁶²³ - Making executor_traits N-nary
- PR #2126¹⁶²⁴ - GCC 4.6 fails to deduce the correct type in lambda
- PR #2125¹⁶²⁵ - Making parcel coalescing test actually test something
- Issue #2124¹⁶²⁶ - Make a testcase for parcel compression
- Issue #2123¹⁶²⁷ - hpx/hpx/runtime/applier_fwd.hpp - Multiple defined types
- Issue #2122¹⁶²⁸ - Exception in primary_namespace::resolve_free_list
- Issue #2121¹⁶²⁹ - Possible memory leak in 1d_stencil_8
- PR #2120¹⁶³⁰ - Fixing 2119
- Issue #2119¹⁶³¹ - reduce_by_key compilation problems
- Issue #2118¹⁶³² - Premature unwrapping of boost::ref’ed arguments
- PR #2117¹⁶³³ - Added missing initializer on last constructor for thread_description

¹⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2141>

¹⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/2139>

¹⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2138>

¹⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2136>

¹⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2135>

¹⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2134>

¹⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2133>

¹⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2132>

¹⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2131>

¹⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2130>

¹⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2129>

¹⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/2128>

¹⁶²³ <https://github.com/STELLAR-GROUP/hpx/pull/2127>

¹⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2126>

¹⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2125>

¹⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2124>

¹⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2123>

¹⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2122>

¹⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2121>

¹⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2120>

¹⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/2119>

¹⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/2118>

¹⁶³³ <https://github.com/STELLAR-GROUP/hpx/pull/2117>

- PR #2116¹⁶³⁴ - Use a lightweight bind implementation when no placeholders are given
- PR #2115¹⁶³⁵ - Replace boost::shared_ptr with std::shared_ptr
- PR #2114¹⁶³⁶ - Adding hook functions for executor_parameter_traits supporting timers
- Issue #2113¹⁶³⁷ - Compilation error with gcc version 4.9.3 (MacPorts gcc49 4.9.3_0)
- PR #2112¹⁶³⁸ - Replace uses of safe_bool with explicit operator bool
- Issue #2111¹⁶³⁹ - Compilation error on QT example
- Issue #2110¹⁶⁴⁰ - Compilation error when passing non-future argument to unwrapped continuation in dataflow
- Issue #2109¹⁶⁴¹ - Warning while compiling hpx
- Issue #2109¹⁶⁴² - Stack trace of last bug causing issues with octotiger
- Issue #2108¹⁶⁴³ - Stack trace of last bug causing issues with octotiger
- PR #2107¹⁶⁴⁴ - Making sure that a missing parcel_coalescing module does not cause startup exceptions
- PR #2106¹⁶⁴⁵ - Stop using hpx_fwd.hpp
- Issue #2105¹⁶⁴⁶ - coalescing plugin handler is not optional any more
- Issue #2104¹⁶⁴⁷ - Make executor_traits N-nary
- Issue #2103¹⁶⁴⁸ - Build error with octotiger and hpx commit e657426d
- PR #2102¹⁶⁴⁹ - Combining thread data storage
- PR #2101¹⁶⁵⁰ - Added repartition version of 1d stencil that uses any performance counter
- PR #2100¹⁶⁵¹ - Drop obsolete TR1 result_of protocol
- PR #2099¹⁶⁵² - Replace uses of boost::bind with util::bind
- PR #2098¹⁶⁵³ - Deprecated inspect checks
- PR #2097¹⁶⁵⁴ - Reduce by key, extends #1141
- PR #2096¹⁶⁵⁵ - Moving local cache from external to hpx/util
- PR #2095¹⁶⁵⁶ - Bump minimum required Boost to 1.50.0

¹⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2116>
¹⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2115>
¹⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2114>
¹⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2113>
¹⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2112>
¹⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2111>
¹⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2110>
¹⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/2109>
¹⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/2109>
¹⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2108>
¹⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2107>
¹⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2106>
¹⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2105>
¹⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2104>
¹⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2103>
¹⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2102>
¹⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2101>
¹⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2100>
¹⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2099>
¹⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2098>
¹⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2097>
¹⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2096>
¹⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2095>

- PR #2094¹⁶⁵⁷ - Add include checks for several Boost utilities
- Issue #2093¹⁶⁵⁸ - ./.../local_cache.hpp(89): error #303: explicit type is missing ("int" assumed)
- PR #2091¹⁶⁵⁹ - Fix for Raspberry pi build
- PR #2090¹⁶⁶⁰ - Fix storage size for util::function<>
- PR #2089¹⁶⁶¹ - Fix #2088
- Issue #2088¹⁶⁶² - More verbose output from cmake configuration
- PR #2087¹⁶⁶³ - Making sure init_globally always executes hpx_main
- Issue #2086¹⁶⁶⁴ - Race condition with recent HPX
- PR #2085¹⁶⁶⁵ - Adding #include checker
- PR #2084¹⁶⁶⁶ - Replace boost lock types with standard library ones
- PR #2083¹⁶⁶⁷ - Simplify packaged task
- PR #2082¹⁶⁶⁸ - Updating APEX version for testing
- PR #2081¹⁶⁶⁹ - Cleanup exception headers
- PR #2080¹⁶⁷⁰ - Make call_once variadic
- Issue #2079¹⁶⁷¹ - With GNU C++, line 85 of hpx/config/version.hpp causes link failure when linking application
- Issue #2078¹⁶⁷² - Simple test fails with _GLIBCXX_DEBUG defined
- PR #2077¹⁶⁷³ - Instantiate board in nqueen client
- PR #2076¹⁶⁷⁴ - Moving coalescing registration to TUs
- PR #2075¹⁶⁷⁵ - Fixed some documentation typos
- PR #2074¹⁶⁷⁶ - Adding flush-mode to message handler flush
- PR #2073¹⁶⁷⁷ - Fixing performance regression introduced lately
- PR #2072¹⁶⁷⁸ - Refactor local::condition_variable
- PR #2071¹⁶⁷⁹ - Timer based on boost::asio::deadline_timer

¹⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2094>

¹⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2093>

¹⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2091>

¹⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2090>

¹⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2089>

¹⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2088>

¹⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2087>

¹⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2086>

¹⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2085>

¹⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2084>

¹⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2083>

¹⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2082>

¹⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2081>

¹⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2080>

¹⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/2079>

¹⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/2078>

¹⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2077>

¹⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2076>

¹⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2075>

¹⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2074>

¹⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2073>

¹⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2072>

¹⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2071>

- PR #2070¹⁶⁸⁰ - Refactor tuple based functionality
- PR #2069¹⁶⁸¹ - Fixed typos
- Issue #2068¹⁶⁸² - Seg fault with octotiger
- PR #2067¹⁶⁸³ - Algorithm cleanup
- PR #2066¹⁶⁸⁴ - Split credit fixes
- PR #2065¹⁶⁸⁵ - Rename HPX_MOVABLE_BUT_NOT_COPYABLE to HPX_MOVABLE_ONLY
- PR #2064¹⁶⁸⁶ - Fixed some typos in docs
- PR #2063¹⁶⁸⁷ - Adding example demonstrating template components
- Issue #2062¹⁶⁸⁸ - Support component templates
- PR #2061¹⁶⁸⁹ - Replace some uses of lexical_cast<string> with C++11 std::to_string
- PR #2060¹⁶⁹⁰ - Replace uses of boost::noncopyable with HPX_NON_COPYABLE
- PR #2059¹⁶⁹¹ - Adding missing for_loop algorithms
- PR #2058¹⁶⁹² - Move several definitions to more appropriate headers
- PR #2057¹⁶⁹³ - Simplify assert_owns_lock and ignore_while_checking
- PR #2056¹⁶⁹⁴ - Replacing std::result_of with util::result_of
- PR #2055¹⁶⁹⁵ - Fix process launching/connecting back
- PR #2054¹⁶⁹⁶ - Add a forwarding coroutine header
- PR #2053¹⁶⁹⁷ - Replace uses of boost::unordered_map with std::unordered_map
- PR #2052¹⁶⁹⁸ - Rewrite tuple unwrap
- PR #2050¹⁶⁹⁹ - Replace uses of BOOST_SCOPED_ENUM with C++11 scoped enums
- PR #2049¹⁷⁰⁰ - Attempt to narrow down split_credit problem
- PR #2048¹⁷⁰¹ - Fixing gcc startup hangs
- PR #2047¹⁷⁰² - Fixing when_xxx and wait_xxx for MSVC12

¹⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2070>
¹⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2069>
¹⁶⁸² <https://github.com/STELLAR-GROUP/hpx/issues/2068>
¹⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2067>
¹⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2066>
¹⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2065>
¹⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2064>
¹⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2063>
¹⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2062>
¹⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2061>
¹⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2060>
¹⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2059>
¹⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2058>
¹⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2057>
¹⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2056>
¹⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2055>
¹⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2054>
¹⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2053>
¹⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2052>
¹⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2050>
¹⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2049>
¹⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2048>
¹⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2047>

- PR #2046¹⁷⁰³ - adding persistent_auto_chunk_size and related tests for for_each
- PR #2045¹⁷⁰⁴ - Fixing HPX_HAVE_THREAD_BACKTRACE_DEPTH build time configuration
- PR #2044¹⁷⁰⁵ - Adding missing service executor types
- PR #2043¹⁷⁰⁶ - Removing ambiguous definitions for is_future_range and future_range_traits
- PR #2042¹⁷⁰⁷ - Clarify that HPX builds can use (much) more than 2GB per process
- PR #2041¹⁷⁰⁸ - Changing future_iterator_traits to support pointers
- Issue #2040¹⁷⁰⁹ - Improve documentation memory usage warning?
- PR #2039¹⁷¹⁰ - Coroutine cleanup
- PR #2038¹⁷¹¹ - Fix cmake policy CMP0042 warning MACOSX_RPATH
- PR #2037¹⁷¹² - Avoid redundant specialization of [unique_]function_nonser
- PR #2036¹⁷¹³ - nvcc dies with an internal error upon pushing/popping warnings inside templates
- Issue #2035¹⁷¹⁴ - Use a less restrictive iterator definition in hpx::lcos::detail::future_iterator_traits
- PR #2034¹⁷¹⁵ - Fixing compilation error with thread queue wait time performance counter
- Issue #2033¹⁷¹⁶ - Compilation error when compiling with thread queue waittime performance counter
- Issue #2032¹⁷¹⁷ - Ambiguous template instantiation for is_future_range and future_range_traits.
- PR #2031¹⁷¹⁸ - Don't restart timer on every incoming parcel
- PR #2030¹⁷¹⁹ - Unify handling of execution policies in parallel algorithms
- PR #2029¹⁷²⁰ - Make pkg-config .pc files use .dylib on OSX
- PR #2028¹⁷²¹ - Adding process component
- PR #2027¹⁷²² - Making check for compiler compatibility independent on compiler path
- PR #2025¹⁷²³ - Fixing inspect tool
- PR #2024¹⁷²⁴ - Intel13 removal
- PR #2023¹⁷²⁵ - Fix errors related to older boost versions and parameter pack expansions in lambdas

¹⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2046>

¹⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2045>

¹⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2044>

¹⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2043>

¹⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2042>

¹⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2041>

¹⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2040>

¹⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2039>

¹⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2038>

¹⁷¹² <https://github.com/STELLAR-GROUP/hpx/pull/2037>

¹⁷¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2036>

¹⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2035>

¹⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2034>

¹⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2033>

¹⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2032>

¹⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2031>

¹⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2030>

¹⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2029>

¹⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2028>

¹⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/2027>

¹⁷²³ <https://github.com/STELLAR-GROUP/hpx/pull/2025>

¹⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2024>

¹⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2023>

- Issue #2022¹⁷²⁶ - gmake fail: “No rule to make target /usr/lib46/libboost_context-mt.so”
- PR #2021¹⁷²⁷ - Added Sudoku example
- Issue #2020¹⁷²⁸ - Make errors related to init_globally.cpp example while building HPX out of the box
- PR #2019¹⁷²⁹ - Fixed some compilation and cmake errors encountered in nqueen example
- PR #2018¹⁷³⁰ - For loop algorithms
- PR #2017¹⁷³¹ - Non-recursive at_index implementation
- Issue #2016¹⁷³² - Add index-based for-loops
- Issue #2015¹⁷³³ - Change default bind-mode to balanced
- PR #2014¹⁷³⁴ - Fixed dataflow if invoked action returns a future
- PR #2013¹⁷³⁵ - Fixing compilation issues with external example
- PR #2012¹⁷³⁶ - Added Sierpinski Triangle example
- Issue #2011¹⁷³⁷ - Compilation error while running sample hello_world_component code
- PR #2010¹⁷³⁸ - Segmented move implemented for hpx::vector
- Issue #2009¹⁷³⁹ - pkg-config order incorrect on 14.04 / GCC 4.8
- Issue #2008¹⁷⁴⁰ - Compilation error in dataflow of action returning a future
- PR #2007¹⁷⁴¹ - Adding new performance counter exposing overall scheduler time
- PR #2006¹⁷⁴² - Function includes
- PR #2005¹⁷⁴³ - Adding an example demonstrating how to initialize HPX from a global object
- PR #2004¹⁷⁴⁴ - Fixing 2000
- PR #2003¹⁷⁴⁵ - Adding generation parameter to gather to enable using it more than once
- PR #2002¹⁷⁴⁶ - Turn on position independent code to solve link problem with hpx_init
- Issue #2001¹⁷⁴⁷ - Gathering more than once segfaults
- Issue #2000¹⁷⁴⁸ - Undefined reference to hpx::assertion_failed

¹⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2022>

¹⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2021>

¹⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2020>

¹⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2019>

¹⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2018>

¹⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2017>

¹⁷³² <https://github.com/STELLAR-GROUP/hpx/issues/2016>

¹⁷³³ <https://github.com/STELLAR-GROUP/hpx/issues/2015>

¹⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2014>

¹⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2013>

¹⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2012>

¹⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2011>

¹⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2010>

¹⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2009>

¹⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2008>

¹⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2007>

¹⁷⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2006>

¹⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2005>

¹⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2004>

¹⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2003>

¹⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2002>

¹⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2001>

¹⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2000>

- [Issue #1999¹⁷⁴⁹](#) - Seg fault in `hpx::lcos::base_lco_with_value<*>::set_value_nonvirt()` when running octo-tiger
- [PR #1998¹⁷⁵⁰](#) - Detect unknown command line options
- [PR #1997¹⁷⁵¹](#) - Extending thread description
- [PR #1996¹⁷⁵²](#) - Adding natvis files to solution (MSVC only)
- [Issue #1995¹⁷⁵³](#) - Command line handling does not produce error
- [PR #1994¹⁷⁵⁴](#) - Possible missing include in `test_utils.hpp`
- [PR #1993¹⁷⁵⁵](#) - Add missing `LANGUAGES` tag to a `hpx_add_compile_flag_if_available()` call in `CMakeLists.txt`
- [PR #1992¹⁷⁵⁶](#) - Fixing `shared_executor_test`
- [PR #1991¹⁷⁵⁷](#) - Making sure the winsock library is properly initialized
- [PR #1990¹⁷⁵⁸](#) - Fixing `bind_test` placeholder ambiguity coming from boost-1.60
- [PR #1989¹⁷⁵⁹](#) - Performance tuning
- [PR #1987¹⁷⁶⁰](#) - Make configurable size of internal storage in `util::function`
- [PR #1986¹⁷⁶¹](#) - AGAS Refactoring+1753 Cache mods
- [PR #1985¹⁷⁶²](#) - Adding missing `task_block::run()` overload taking an executor
- [PR #1984¹⁷⁶³](#) - Adding an optimized LRU Cache implementation (for AGAS)
- [PR #1983¹⁷⁶⁴](#) - Avoid invoking migration table look up for all objects
- [PR #1981¹⁷⁶⁵](#) - Replacing `uintptr_t` (which is not defined everywhere) with `std::size_t`
- [PR #1980¹⁷⁶⁶](#) - Optimizing LCO continuations
- [PR #1979¹⁷⁶⁷](#) - Fixing Cori
- [PR #1978¹⁷⁶⁸](#) - Fix test check that got broken in hasty fix to memory overflow
- [PR #1977¹⁷⁶⁹](#) - Refactor action traits
- [PR #1976¹⁷⁷⁰](#) - Fixes typo in `README.rst`
- [PR #1975¹⁷⁷¹](#) - Reduce size of benchmark timing arrays to fix test failures

¹⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1999>

¹⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1998>

¹⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1997>

¹⁷⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1996>

¹⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1995>

¹⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1994>

¹⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1993>

¹⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1992>

¹⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1991>

¹⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1990>

¹⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1989>

¹⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1987>

¹⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1986>

¹⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1985>

¹⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1984>

¹⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1983>

¹⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1981>

¹⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1980>

¹⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1979>

¹⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1978>

¹⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1977>

¹⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1976>

¹⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1975>

- PR #1974¹⁷⁷² - Add action to update data owned by the partitioned_vector component
- PR #1972¹⁷⁷³ - Adding partitioned_vector SPMD example
- PR #1971¹⁷⁷⁴ - Fixing 1965
- PR #1970¹⁷⁷⁵ - Papi fixes
- PR #1969¹⁷⁷⁶ - Fixing continuation recursions to not depend on fixed amount of recursions
- PR #1968¹⁷⁷⁷ - More segmented algorithms
- Issue #1967¹⁷⁷⁸ - Simplify component implementations
- PR #1966¹⁷⁷⁹ - Migrate components
- Issue #1964¹⁷⁸⁰ - fatal error: 'boost/lockfree/detail/branch_hints.hpp' file not found
- Issue #1962¹⁷⁸¹ - parallel::copy_if has race condition when used on in place arrays
- PR #1963¹⁷⁸² - Fixing Static Parcelport initialization
- PR #1961¹⁷⁸³ - Fix function target
- Issue #1960¹⁷⁸⁴ - Papi counters don't reset
- PR #1959¹⁷⁸⁵ - Fixing 1958
- Issue #1958¹⁷⁸⁶ - inclusive_scan gives incorrect results with non-commutative operator
- PR #1957¹⁷⁸⁷ - Fixing #1950
- PR #1956¹⁷⁸⁸ - Sort by key example
- PR #1955¹⁷⁸⁹ - Adding regression test for #1946: Hang in wait_all() in distributed run
- Issue #1954¹⁷⁹⁰ - HPX releases should not use -Werror
- PR #1953¹⁷⁹¹ - Adding performance analysis for AGAS cache
- PR #1952¹⁷⁹² - Adapting test for explicit variadics to fail for gcc 4.6
- PR #1951¹⁷⁹³ - Fixing memory leak
- Issue #1950¹⁷⁹⁴ - Simplify external builds

¹⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1974>

¹⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1972>

¹⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1971>

¹⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1970>

¹⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1969>

¹⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1968>

¹⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1967>

¹⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1966>

¹⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1964>

¹⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1962>

¹⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1963>

¹⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1961>

¹⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1960>

¹⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1959>

¹⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1958>

¹⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1957>

¹⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1956>

¹⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1955>

¹⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1954>

¹⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1953>

¹⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1952>

¹⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1951>

¹⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1950>

- [PR #1949¹⁷⁹⁵](#) - Fixing yet another lock that is being held during suspension
- [PR #1948¹⁷⁹⁶](#) - Fixed container algorithms for Intel
- [PR #1947¹⁷⁹⁷](#) - Adding workaround for tagged_tuple
- [Issue #1946¹⁷⁹⁸](#) - Hang in wait_all() in distributed run
- [PR #1945¹⁷⁹⁹](#) - Fixed container algorithm tests
- [Issue #1944¹⁸⁰⁰](#) - assertion 'p.destination_locality() == hpx::get_locality()' failed
- [PR #1943¹⁸⁰¹](#) - Fix a couple of compile errors with clang
- [PR #1942¹⁸⁰²](#) - Making parcel coalescing functional
- [Issue #1941¹⁸⁰³](#) - Re-enable parcel coalescing
- [PR #1940¹⁸⁰⁴](#) - Touching up make_future
- [PR #1939¹⁸⁰⁵](#) - Fixing problems in over-subscription management in the resource manager
- [PR #1938¹⁸⁰⁶](#) - Removing use of unified Boost.Thread header
- [PR #1937¹⁸⁰⁷](#) - Cleaning up the use of Boost.Accumulator headers
- [PR #1936¹⁸⁰⁸](#) - Making sure interval timer is started for aggregating performance counters
- [PR #1935¹⁸⁰⁹](#) - Tagged results
- [PR #1934¹⁸¹⁰](#) - Fix remote async with deferred launch policy
- [Issue #1933¹⁸¹¹](#) - Floating point exception in `statistics_counter<boost::accumulators::tag::mean>::get_c`
- [PR #1932¹⁸¹²](#) - Removing superfluous includes of boost/lockfree/detail/branch_hints.hpp
- [PR #1931¹⁸¹³](#) - fix compilation with clang 3.8.0
- [Issue #1930¹⁸¹⁴](#) - Missing online documentation for HPX 0.9.11
- [PR #1929¹⁸¹⁵](#) - LWG2485: get() should be overloaded for const tuple&&
- [PR #1928¹⁸¹⁶](#) - Revert “Using ninja for circle-ci builds”
- [PR #1927¹⁸¹⁷](#) - Using ninja for circle-ci builds

¹⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1949>

¹⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1948>

¹⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1947>

¹⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1946>

¹⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1945>

¹⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1944>

¹⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1943>

¹⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1942>

¹⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1941>

¹⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1940>

¹⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1939>

¹⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1938>

¹⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1937>

¹⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1936>

¹⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1935>

¹⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1934>

¹⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1933>

¹⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/1932>

¹⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1931>

¹⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1930>

¹⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1929>

¹⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1928>

¹⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1927>

- PR #1926¹⁸¹⁸ - Fixing serialization of `std::array`
- Issue #1925¹⁸¹⁹ - Issues with static HPX libraries
- Issue #1924¹⁸²⁰ - Performance degrading over time
- Issue #1923¹⁸²¹ - serialization of `std::array` appears broken in latest commit
- PR #1922¹⁸²² - Container algorithms
- PR #1921¹⁸²³ - Tons of smaller quality improvements
- Issue #1920¹⁸²⁴ - Seg fault in `hpx::serialization::output_archive::add_gid` when running octotiger
- Issue #1919¹⁸²⁵ - Intel 15 compiler bug preventing HPX build
- PR #1918¹⁸²⁶ - Address sanitizer fixes
- PR #1917¹⁸²⁷ - Fixing compilation problems of `parallel::sort` with Intel compilers
- PR #1916¹⁸²⁸ - Making sure code compiles if `HPX_WITH_HWLOC=Off`
- Issue #1915¹⁸²⁹ - `max_cores` undefined if `HPX_WITH_HWLOC=Off`
- PR #1913¹⁸³⁰ - Add utility member functions for `partitioned_vector`
- PR #1912¹⁸³¹ - Adding support for invoking actions to dataflow
- PR #1911¹⁸³² - Adding first batch of container algorithms
- PR #1910¹⁸³³ - Keep `cmake_module_path`
- PR #1909¹⁸³⁴ - Fix `mpirun` with `pbs`
- PR #1908¹⁸³⁵ - Changing `parallel::sort` to return the last iterator as proposed by N4560
- PR #1907¹⁸³⁶ - Adding a minimum version for Open MPI
- PR #1906¹⁸³⁷ - Updates to the Release Procedure
- PR #1905¹⁸³⁸ - Fixing #1903
- PR #1904¹⁸³⁹ - Making sure `std` containers are cleared before serialization loads data
- Issue #1903¹⁸⁴⁰ - When running octotiger, I get: `assertion '(*new_gids_)[gid].size() == 1'`
failed: `HPX(assertion_failure)`

¹⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1926>
¹⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1925>
¹⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1924>
¹⁸²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1923>
¹⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/1922>
¹⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/1921>
¹⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1920>
¹⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1919>
¹⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1918>
¹⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1917>
¹⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1916>
¹⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1915>
¹⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1913>
¹⁸³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1912>
¹⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/1911>
¹⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/1910>
¹⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1909>
¹⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1908>
¹⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1907>
¹⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1906>
¹⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1905>
¹⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1904>
¹⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1903>

- [Issue #1902](#)¹⁸⁴¹ - Immediate crash when running hpx/octotiger with `_GLIBCXX_DEBUG` defined.
- [PR #1901](#)¹⁸⁴² - Making non-serializable classes non-serializable
- [Issue #1900](#)¹⁸⁴³ - Two possible issues with `std::list` serialization
- [PR #1899](#)¹⁸⁴⁴ - Fixing a problem with credit splitting as revealed by #1898
- [Issue #1898](#)¹⁸⁴⁵ - Accessing component from locality where it was not created segfaults
- [PR #1897](#)¹⁸⁴⁶ - Changing `parallel::sort` to return the last iterator as proposed by N4560
- [Issue #1896](#)¹⁸⁴⁷ - version 1.0?
- [Issue #1895](#)¹⁸⁴⁸ - Warning comment on `numa_allocator` is not very clear
- [PR #1894](#)¹⁸⁴⁹ - Add support for compilers that have `thread_local`
- [PR #1893](#)¹⁸⁵⁰ - Fixing 1890
- [PR #1892](#)¹⁸⁵¹ - Adds typed `future_type` for `executor_traits`
- [PR #1891](#)¹⁸⁵² - Fix wording in certain parallel algorithm docs
- [Issue #1890](#)¹⁸⁵³ - Invoking papi counters give segfault
- [PR #1889](#)¹⁸⁵⁴ - Fixing problems as reported by clang-check
- [PR #1888](#)¹⁸⁵⁵ - WIP parallel `is_heap`
- [PR #1887](#)¹⁸⁵⁶ - Fixed resetting performance counters related to idle-rate, etc
- [Issue #1886](#)¹⁸⁵⁷ - Run hpx with `qsub` does not work
- [PR #1885](#)¹⁸⁵⁸ - Warning cleaning pass
- [PR #1884](#)¹⁸⁵⁹ - Add missing parallel algorithm header
- [PR #1883](#)¹⁸⁶⁰ - Add feature test for `thread_local` on Clang for TLS
- [PR #1882](#)¹⁸⁶¹ - Fix some redundant qualifiers
- [Issue #1881](#)¹⁸⁶² - Unable to compile Octotiger using HPX and Intel MPI on SuperMIC
- [Issue #1880](#)¹⁸⁶³ - clang with `libc++` on Linux needs TLS case

¹⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1902>

¹⁸⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1901>

¹⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1900>

¹⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1899>

¹⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1898>

¹⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1897>

¹⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1896>

¹⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1895>

¹⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1894>

¹⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1893>

¹⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1892>

¹⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1891>

¹⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1890>

¹⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1889>

¹⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1888>

¹⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1887>

¹⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1886>

¹⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1885>

¹⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1884>

¹⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1883>

¹⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1882>

¹⁸⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1881>

¹⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1880>

- PR #1879¹⁸⁶⁴ - Doc fixes for #1868
- PR #1878¹⁸⁶⁵ - Simplify functions
- PR #1877¹⁸⁶⁶ - Removing most usage of Boost.Config
- PR #1876¹⁸⁶⁷ - Add missing parallel algorithms to algorithm.hpp
- PR #1875¹⁸⁶⁸ - Simplify callables
- PR #1874¹⁸⁶⁹ - Address long standing FIXME on using `std::unique_ptr` with incomplete types
- PR #1873¹⁸⁷⁰ - Fixing 1871
- PR #1872¹⁸⁷¹ - Making sure PBS environment uses specified node list even if no PBS_NODEFILE env is available
- Issue #1871¹⁸⁷² - Fortran checks should be optional
- PR #1870¹⁸⁷³ - Touch local::mutex
- PR #1869¹⁸⁷⁴ - Documentation refactoring based off #1868
- PR #1867¹⁸⁷⁵ - Embrace static_assert
- PR #1866¹⁸⁷⁶ - Fix #1803 with documentation refactoring
- PR #1865¹⁸⁷⁷ - Setting OUTPUT_NAME as target properties
- PR #1863¹⁸⁷⁸ - Use SYSTEM for boost includes
- PR #1862¹⁸⁷⁹ - Minor cleanups
- PR #1861¹⁸⁸⁰ - Minor Corrections for Release
- PR #1860¹⁸⁸¹ - Fixing hpx gdb script
- Issue #1859¹⁸⁸² - reset_active_counters resets times and thread counts before some of the counters are evaluated
- PR #1858¹⁸⁸³ - Release V0.9.11
- PR #1857¹⁸⁸⁴ - removing diskperf example from 9.11 release
- PR #1856¹⁸⁸⁵ - fix return in packaged_task_base::reset()
- Issue #1842¹⁸⁸⁶ - Install error: file INSTALL cannot find libhpx_parcel_coalescing.so.0.9.11

¹⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1879>
¹⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1878>
¹⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1877>
¹⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1876>
¹⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1875>
¹⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1874>
¹⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1873>
¹⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1872>
¹⁸⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1871>
¹⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1870>
¹⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1869>
¹⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1867>
¹⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1866>
¹⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1865>
¹⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1863>
¹⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1862>
¹⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1861>
¹⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1860>
¹⁸⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1859>
¹⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1858>
¹⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1857>
¹⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1856>
¹⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1842>

- PR #1839¹⁸⁸⁷ - Adding fedora docs
- PR #1824¹⁸⁸⁸ - Changing version on master to V0.9.12
- PR #1818¹⁸⁸⁹ - Fixing #1748
- Issue #1815¹⁸⁹⁰ - seg fault in AGAS
- Issue #1803¹⁸⁹¹ - wait_all documentation
- Issue #1796¹⁸⁹² - Outdated documentation to be revised
- Issue #1759¹⁸⁹³ - glibc munmap_chunk or free(): invalid pointer on SuperMIC
- Issue #1753¹⁸⁹⁴ - HPX performance degrades with time since execution begins
- Issue #1748¹⁸⁹⁵ - All public HPX headers need to be self contained
- PR #1719¹⁸⁹⁶ - How to build HPX with Visual Studio
- Issue #1684¹⁸⁹⁷ - Race condition when using -hpx:connect?
- PR #1658¹⁸⁹⁸ - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1641¹⁸⁹⁹ - Generic client
- Issue #1632¹⁹⁰⁰ - heartbeat example fails on separate nodes
- PR #1603¹⁹⁰¹ - Adds preferred namespace check to inspect tool
- Issue #1559¹⁹⁰² - Extend inspect tool
- Issue #1523¹⁹⁰³ - Remote async with deferred launch policy never executes
- Issue #1472¹⁹⁰⁴ - Serialization issues
- Issue #1457¹⁹⁰⁵ - Implement N4392: C++ Latches and Barriers
- PR #1444¹⁹⁰⁶ - Enabling usage of moveonly types for component construction
- Issue #1407¹⁹⁰⁷ - The Intel 13 compiler has failing unit tests
- Issue #1405¹⁹⁰⁸ - Allow component constructors to take movable only types
- Issue #1265¹⁹⁰⁹ - Enable dataflow() to be usable with actions

¹⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1839>

¹⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1824>

¹⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1818>

¹⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1815>

¹⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1803>

¹⁸⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1796>

¹⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1759>

¹⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1753>

¹⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1748>

¹⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1719>

¹⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1684>

¹⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1658>

¹⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1641>

¹⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1632>

¹⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1603>

¹⁹⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1559>

¹⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1523>

¹⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1472>

¹⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1457>

¹⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1444>

¹⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1407>

¹⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1405>

¹⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1265>

- [Issue #1236](#)¹⁹¹⁰ - NUMA aware allocators
- [Issue #802](#)¹⁹¹¹ - Fix Broken Examples
- [Issue #559](#)¹⁹¹² - Add `hpx::migrate` facility
- [Issue #449](#)¹⁹¹³ - Make actions with template arguments usable and add documentation
- [Issue #279](#)¹⁹¹⁴ - Refactor `addressing_service` into a base class and two derived classes
- [Issue #224](#)¹⁹¹⁵ - Changing thread state metadata is not thread safe
- [Issue #55](#)¹⁹¹⁶ - 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 choice of either using one of the predefined executor types or to provide its own, possibly application specific, executor. We paid very close attention to align all of these changes with the existing C++ Standards documents or with the ongoing proposals for standardization.

This release is the first after our change to a new development policy. We switched all development to be strictly performed on branches only, all direct commits to our main branch (`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](#)¹⁹¹⁷ hosted continuous integration service. This gives us constant and immediate feedback on the health of our master branch.
- The compiler configuration subsystem in the build system has been reimplemented. Instead of using `Boost.Config` we now use our own lightweight set of `cmake` scripts to determine the available language and library features supported by the used compiler.
- The API for creating instances of components has been consolidated. All component instances should be created using the `hpx::new_` only. It allows 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_`.

¹⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1236>

¹⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/802>

¹⁹¹² <https://github.com/STELLAR-GROUP/hpx/issues/559>

¹⁹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/449>

¹⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/279>

¹⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/224>

¹⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/55>

¹⁹¹⁷ <https://circleci.com/gh/STELLAR-GROUP/hpx>

- 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 distribution policy.
- The new distribution policies can now be also used with `hpx::async`. This change also deprecates `hpx::async_colocated(id, ...)` which now is replaced by a distribution policy: `hpx::async(hpx::colocated(id), ...)`.
- The `hpx::vector` and `hpx::unordered_map` data structures can now be used with the new distribution policies as well.
- The parallel facility `hpx::parallel::task_region` has been renamed to `hpx::parallel::task_block` based on the changes in the corresponding standardization proposal N4411¹⁹¹⁸.
- 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 N4399¹⁹¹⁹).
- Added performance counters exposing data related to data transferred by input/output (filesystem) operations (thanks to Maciej Brodowicz).
- Added performance counters allowing to track the number of action invocations (local and remote invocations).
- Added new command line options `-hpx:print-counter-at` and `-hpx:reset-counters`.
- The `hpx::vector` component has been renamed to `hpx::partitioned_vector` to make it explicit that the underlying memory is not contiguous.
- Introduced a completely new and uniform higher-level parallelism API which is based on executors. All existing parallelism APIs have been adapted to this. We have added a large number of different executor types, such as a numa-aware executor, a this-thread executor, etc.
- Added support for the MingW toolchain on Windows (thanks to Eric Lemanissier).
- HPX now includes support for APEX, (Autonomic Performance Environment for eXascale). APEX is an instrumentation and software adaptation library that provides an interface to TAU profiling / tracing as well as runtime adaptation of HPX applications through policy definitions. For more information and documentation, please see <https://github.com/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 #1141¹⁹²⁰ for the list of all available parallel algorithms (thanks to Daniel Bourgeois and John Biddiscombe for contributing their work).

¹⁹¹⁸ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

¹⁹¹⁹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html>

¹⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1141>

Breaking changes

- We are moving into the direction of unifying managed and simple components. In order to stop exposing the old facilities, all examples have been converted to use the new classes. The breaking change in this release is that performance counters are now a `hpx::components::component_base` instead of `hpx::components::managed_component_base`.
- We removed the support for stackless threads. It turned out that there was no performance benefit when using stackless threads. As such, we decided to clean up our codebase. This feature was not documented.
- The CMake project name has changed from 'hpx' to 'HPX' for consistency and compatibility with naming conventions and other CMake projects. Generated config files go into `<prefix>/lib/cmake/HPX` and not `<prefix>/lib/cmake/hpx`.
- The macro `HPX_REGISTER_MINIMAL_COMPONENT_FACTORY` has been deprecated. Please use `HPX_REGISTER_COMPONENT` instead. The old macro will be removed in the next release.
- The obsolete `distributing_factory` and `binpacking_factory` components have been removed. The corresponding functionality is now provided by the `hpx::new_` API function in conjunction with the `hpx::default_layout` and `hpx::binpacking` distribution policies (`hpx::components::default_distribution_policy` and `hpx::components::binpacking_distribution_policy`).
- The API function `hpx::new_colocated` has been deprecated. Please use the consolidated API `hpx::new_` in conjunction with the new `hpx::colocated` distribution policy (`hpx::components::colocating_distribution_policy`) instead. The old API function will still be available for at least one release of *HPX* if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The API function `hpx::async_colocated` has been deprecated. Please use the consolidated API `hpx::async` in conjunction with the new `hpx::colocated` distribution policy (`hpx::components::colocating_distribution_policy`) instead. The old API function will still be available for at least one release of *HPX* if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The obsolete `remote_object` component has been removed.
- Replaced the use of `Boost.Serialization` with our own solution. While the new version is mostly compatible with `Boost.Serialization`, this change requires some minor code modifications in user code. For more information, please see the corresponding [announcement](#)¹⁹²¹ 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 as `-DHPX_WITH_MALLOC=...` and will cause the preprocessor constant `HPX_HAVE_MALLOC` to be defined. The actual name of the constant (i.e. `MALLOC`) has not changed. Please see the corresponding documentation for more details (*CMake variables used to configure HPX*).
- The `get_gid()` functions exposed by the component base classes `hpx::components::server::simple_component_base`, `hpx::components::server::managed_component_base` and `hpx::components::server::fixed_component_base` have been replaced by two new functions: `get_unmanaged_id()` and `get_id()`. To enable the old function name for backwards compatibility, use the cmake configuration option `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY=On`.
- All functions which were named `get_gid()` but were returning `hpx::id_type` have been renamed to `get_id()`. To enable the old function names for backwards compatibility, use the cmake configuration option `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY=On`.

¹⁹²¹ <http://thread.gmane.org/gmane.comp.lib.hpx.devel/196>

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #1855¹⁹²² - Completely removing external/endian
- PR #1854¹⁹²³ - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1853¹⁹²⁴ - Updating CMake configuration to get correct version of TAU library
- PR #1852¹⁹²⁵ - Fixing Performance Problems with MPI Parcelport
- PR #1851¹⁹²⁶ - Fixing hpx_add_link_flag() and hpx_remove_link_flag()
- PR #1850¹⁹²⁷ - Fixing 1836, adding parallel::sort
- PR #1849¹⁹²⁸ - Fixing configuration for use of more than 64 cores
- PR #1848¹⁹²⁹ - Change default APEX version for release
- PR #1847¹⁹³⁰ - Fix client_base::then on release
- PR #1846¹⁹³¹ - Removing broken lcos::local::channel from release
- PR #1845¹⁹³² - Adding example demonstrating a possible safe-object implementation to release
- PR #1844¹⁹³³ - Removing stubs from accumulator examples
- PR #1843¹⁹³⁴ - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1841¹⁹³⁵ - Fixing client_base<>::then
- PR #1840¹⁹³⁶ - Adding example demonstrating a possible safe-object implementation
- PR #1838¹⁹³⁷ - Update version rc1
- PR #1837¹⁹³⁸ - Removing broken lcos::local::channel
- PR #1835¹⁹³⁹ - Adding explicit move constructor and assignment operator to hpx::lcos::promise
- PR #1834¹⁹⁴⁰ - Making hpx::lcos::promise move-only
- PR #1833¹⁹⁴¹ - Adding fedora docs
- Issue #1832¹⁹⁴² - hpx::lcos::promise<> must be move-only

¹⁹²² <https://github.com/STELLAR-GROUP/hpx/pull/1855>

¹⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/1854>

¹⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1853>

¹⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1852>

¹⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1851>

¹⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1850>

¹⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1849>

¹⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1848>

¹⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1847>

¹⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1846>

¹⁹³² <https://github.com/STELLAR-GROUP/hpx/pull/1845>

¹⁹³³ <https://github.com/STELLAR-GROUP/hpx/pull/1844>

¹⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1843>

¹⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1841>

¹⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1840>

¹⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1838>

¹⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1837>

¹⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1835>

¹⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1834>

¹⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1833>

¹⁹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1832>

- PR #1831¹⁹⁴³ - Fixing resource manager gcc5.2
- PR #1830¹⁹⁴⁴ - Fix intel13
- PR #1829¹⁹⁴⁵ - Unbreaking thread test
- PR #1828¹⁹⁴⁶ - Fixing #1620
- PR #1827¹⁹⁴⁷ - Fixing a memory management issue for the Parquet application
- Issue #1826¹⁹⁴⁸ - Memory management issue in `hpx::lcos::promise`
- PR #1825¹⁹⁴⁹ - Adding `hpx::components::component` and `hpx::components::component_base`
- PR #1823¹⁹⁵⁰ - Adding git commit id to circleci build
- PR #1822¹⁹⁵¹ - applying fixes suggested by clang 3.7
- PR #1821¹⁹⁵² - Hyperlink fixes
- PR #1820¹⁹⁵³ - added parallel multi-locality sanity test
- PR #1819¹⁹⁵⁴ - Fixing #1667
- Issue #1817¹⁹⁵⁵ - Hyperlinks generated by inspect tool are wrong
- PR #1816¹⁹⁵⁶ - Support `hpxrx`
- PR #1814¹⁹⁵⁷ - Fix `async` to dispatch to the correct locality in all cases
- Issue #1813¹⁹⁵⁸ - `async(launch::..., action(), ...)` always invokes locally
- PR #1812¹⁹⁵⁹ - fixed syntax error in `CMakeLists.txt`
- PR #1811¹⁹⁶⁰ - Agas optimizations
- PR #1810¹⁹⁶¹ - drop superfluous typedefs
- PR #1809¹⁹⁶² - Allow HPX to be used as an optional package in 3rd party code
- PR #1808¹⁹⁶³ - Fixing #1723
- PR #1807¹⁹⁶⁴ - Making sure `resolve_localities` does not hang during normal operation
- Issue #1806¹⁹⁶⁵ - Spinlock no longer movable and deletes operator `'='`, breaks MiniGhost

¹⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1831>

¹⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1830>

¹⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1829>

¹⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1828>

¹⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1827>

¹⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1826>

¹⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1825>

¹⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1823>

¹⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1822>

¹⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1821>

¹⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1820>

¹⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1819>

¹⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1817>

¹⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1816>

¹⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1814>

¹⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1813>

¹⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1812>

¹⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1811>

¹⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1810>

¹⁹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1809>

¹⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1808>

¹⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1807>

¹⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1806>

- Issue #1804¹⁹⁶⁶ - register_with_basename causes hangs
- PR #1801¹⁹⁶⁷ - Enhanced the inspect tool to take user directly to the problem with hyperlinks
- Issue #1800¹⁹⁶⁸ - Problems compiling application on smic
- PR #1799¹⁹⁶⁹ - Fixing cv exceptions
- PR #1798¹⁹⁷⁰ - Documentation refactoring & updating
- PR #1797¹⁹⁷¹ - Updating the activeharmony CMake module
- PR #1795¹⁹⁷² - Fixing cv
- PR #1794¹⁹⁷³ - Fix connect with hpx::runtime_mode_connect
- PR #1793¹⁹⁷⁴ - fix a wrong use of HPX_MAX_CPU_COUNT instead of HPX_HAVE_MAX_CPU_COUNT
- PR #1792¹⁹⁷⁵ - Allow for default constructed parcel instances to be moved
- PR #1791¹⁹⁷⁶ - Fix connect with hpx::runtime_mode_connect
- Issue #1790¹⁹⁷⁷ - assertion action_.get() failed: HPX(assertion_failure) when running Octotiger with pull request 1786
- PR #1789¹⁹⁷⁸ - Fixing discover_counter_types API function
- Issue #1788¹⁹⁷⁹ - connect with hpx::runtime_mode_connect
- Issue #1787¹⁹⁸⁰ - discover_counter_types not working
- PR #1786¹⁹⁸¹ - Changing addressing_service to use std::unordered_map instead of std::map
- PR #1785¹⁹⁸² - Fix is_iterator for container algorithms
- PR #1784¹⁹⁸³ - Adding new command line options:
- PR #1783¹⁹⁸⁴ - Minor changes for APEX support
- PR #1782¹⁹⁸⁵ - Drop legacy forwarding action traits
- PR #1781¹⁹⁸⁶ - Attempt to resolve the race between cv::wait_xxx and cv::notify_all
- PR #1780¹⁹⁸⁷ - Removing serialize_sequence
- PR #1779¹⁹⁸⁸ - Fixed #1501: hwloc configuration options are wrong for MIC

¹⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1804>

¹⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1801>

¹⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1800>

¹⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1799>

¹⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1798>

¹⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1797>

¹⁹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1795>

¹⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1794>

¹⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1793>

¹⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1792>

¹⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1791>

¹⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1790>

¹⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1789>

¹⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1788>

¹⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1787>

¹⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1786>

¹⁹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1785>

¹⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1784>

¹⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1783>

¹⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1782>

¹⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1781>

¹⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1780>

¹⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1779>

- PR #1778¹⁹⁸⁹ - Removing ability to enable/disable parcel handling
- PR #1777¹⁹⁹⁰ - Completely removing stackless threads
- PR #1776¹⁹⁹¹ - Cleaning up util/plugin
- PR #1775¹⁹⁹² - Agas fixes
- PR #1774¹⁹⁹³ - Action invocation count
- PR #1773¹⁹⁹⁴ - replaced MSVC variable with WIN32
- PR #1772¹⁹⁹⁵ - Fixing Problems in MPI parcellport and future serialization.
- PR #1771¹⁹⁹⁶ - Fixing intel 13 compiler errors related to variadic template template parameters for `lcos::when_` tests
- PR #1770¹⁹⁹⁷ - Forwarding decay to `std::`
- PR #1769¹⁹⁹⁸ - Add more characters with special regex meaning to the existing patch
- PR #1768¹⁹⁹⁹ - Adding test for `receive_buffer`
- PR #1767²⁰⁰⁰ - Making sure that uptime counter throws exception on any attempt to be reset
- PR #1766²⁰⁰¹ - Cleaning up code related to throttling scheduler
- PR #1765²⁰⁰² - Restricting `thread_data` to creating only with `intrusive_pointers`
- PR #1764²⁰⁰³ - Fixing 1763
- Issue #1763²⁰⁰⁴ - UB in `thread_data::operator delete`
- PR #1762²⁰⁰⁵ - Making sure all serialization registries/factories are unique
- PR #1761²⁰⁰⁶ - Fixed #1751: `hpx::future::wait_for` fails a simple test
- PR #1758²⁰⁰⁷ - Fixing #1757
- Issue #1757²⁰⁰⁸ - pinning not correct using `-hpx:bind`
- Issue #1756²⁰⁰⁹ - compilation error with MinGW
- PR #1755²⁰¹⁰ - Making output serialization const-correct
- Issue #1753²⁰¹¹ - HPX performance degrades with time since execution begins

¹⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1778>

¹⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1777>

¹⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1776>

¹⁹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1775>

¹⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1774>

¹⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1773>

¹⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1772>

¹⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1771>

¹⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1770>

¹⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1769>

¹⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1768>

²⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1767>

²⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1766>

²⁰⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1765>

²⁰⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1764>

²⁰⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1763>

²⁰⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1762>

²⁰⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1761>

²⁰⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1758>

²⁰⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1757>

²⁰⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1756>

²⁰¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1755>

²⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1753>

- [Issue #1752²⁰¹²](#) - Error in AGAS
- [Issue #1751²⁰¹³](#) - `hpx::future::wait_for` fails a simple test
- [PR #1750²⁰¹⁴](#) - Removing `hpx_fwd.hpp` includes
- [PR #1749²⁰¹⁵](#) - Simplify `result_of` and friends
- [PR #1747²⁰¹⁶](#) - Removed superfluous code from `message_buffer.hpp`
- [PR #1746²⁰¹⁷](#) - Tuple dependencies
- [Issue #1745²⁰¹⁸](#) - Broken when `_some` which takes iterators
- [PR #1744²⁰¹⁹](#) - Refining archive interface
- [PR #1743²⁰²⁰](#) - Fixing when `_all` when only a single future is passed
- [PR #1742²⁰²¹](#) - Config includes
- [PR #1741²⁰²²](#) - Os executors
- [Issue #1740²⁰²³](#) - `hpx::promise` has some problems
- [PR #1739²⁰²⁴](#) - Parallel composition with generic containers
- [Issue #1738²⁰²⁵](#) - After building program and successfully linking to a version of `hpx` `DHPX_DIR` seems to be ignored
- [Issue #1737²⁰²⁶](#) - Uptime problems
- [PR #1736²⁰²⁷](#) - added convenience `c-tor` and `begin()/end()` to `serialize_buffer`
- [PR #1735²⁰²⁸](#) - Config includes
- [PR #1734²⁰²⁹](#) - Fixed #1688: Add timer counters for `tfunc_total` and `exec_total`
- [Issue #1733²⁰³⁰](#) - Add unit test for `hpx/lcos/local/receive_buffer.hpp`
- [PR #1732²⁰³¹](#) - Renaming `get_os_thread_count`
- [PR #1731²⁰³²](#) - Basename registration
- [Issue #1730²⁰³³](#) - Use after move of `thread_init_data`
- [PR #1729²⁰³⁴](#) - Rewriting channel based on new gate component

²⁰¹² <https://github.com/STELLAR-GROUP/hpx/issues/1752>

²⁰¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1751>

²⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1750>

²⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1749>

²⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1747>

²⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1746>

²⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1745>

²⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1744>

²⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1743>

²⁰²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1742>

²⁰²² <https://github.com/STELLAR-GROUP/hpx/pull/1741>

²⁰²³ <https://github.com/STELLAR-GROUP/hpx/issues/1740>

²⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1739>

²⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1738>

²⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1737>

²⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1736>

²⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1735>

²⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1734>

²⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1733>

²⁰³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1732>

²⁰³² <https://github.com/STELLAR-GROUP/hpx/pull/1731>

²⁰³³ <https://github.com/STELLAR-GROUP/hpx/issues/1730>

²⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1729>

- PR #1728²⁰³⁵ - Fixing #1722
- PR #1727²⁰³⁶ - Fixing compile problems with `apply_collocated`
- PR #1726²⁰³⁷ - Apex integration
- PR #1725²⁰³⁸ - fixed test timeouts
- PR #1724²⁰³⁹ - Renaming vector
- Issue #1723²⁰⁴⁰ - Drop support for intel compilers and gcc 4.4. based standard libs
- Issue #1722²⁰⁴¹ - Add support for detecting non-ready futures before serialization
- PR #1721²⁰⁴² - Unifying parallel executors, initializing from launch policy
- PR #1720²⁰⁴³ - dropped superfluous typedef
- Issue #1718²⁰⁴⁴ - Windows 10 x64, VS 2015 - Unknown CMake command “`add_hpx_pseudo_target`”.
- PR #1717²⁰⁴⁵ - Timed executor traits for thread-executors
- PR #1716²⁰⁴⁶ - serialization of arrays didn’t work with non-pod types. fixed
- PR #1715²⁰⁴⁷ - List serialization
- PR #1714²⁰⁴⁸ - changing misspellings
- PR #1713²⁰⁴⁹ - Fixed distribution policy executors
- PR #1712²⁰⁵⁰ - Moving library detection to be executed after feature tests
- PR #1711²⁰⁵¹ - Simplify parcel
- PR #1710²⁰⁵² - Compile only tests
- PR #1709²⁰⁵³ - Implemented timed executors
- PR #1708²⁰⁵⁴ - Implement `parallel::executor_traits` for thread-executors
- PR #1707²⁰⁵⁵ - Various fixes to `threads::executors` to make custom schedulers work
- PR #1706²⁰⁵⁶ - Command line option `-hpx:cores` does not work as expected
- Issue #1705²⁰⁵⁷ - command line option `-hpx:cores` does not work as expected

²⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1728>

²⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1727>

²⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1726>

²⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1725>

²⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1724>

²⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1723>

²⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1722>

²⁰⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1721>

²⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1720>

²⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1718>

²⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1717>

²⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1716>

²⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1715>

²⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1714>

²⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1713>

²⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1712>

²⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1711>

²⁰⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1710>

²⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1709>

²⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1708>

²⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1707>

²⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1706>

²⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1705>

- PR #1704²⁰⁵⁸ - vector deserialization is speeded up a little
- PR #1703²⁰⁵⁹ - Fixing shared_mutes
- Issue #1702²⁰⁶⁰ - Shared_mutex does not compile with no_mutex cond_var
- PR #1701²⁰⁶¹ - Add distribution_policy_executor
- PR #1700²⁰⁶² - Executor parameters
- PR #1699²⁰⁶³ - Readers writer lock
- PR #1698²⁰⁶⁴ - Remove leftovers
- PR #1697²⁰⁶⁵ - Fixing held locks
- PR #1696²⁰⁶⁶ - Modified Scan Partitioner for Algorithms
- PR #1695²⁰⁶⁷ - This thread executors
- PR #1694²⁰⁶⁸ - Fixed #1688: Add timer counters for tfunc_total and exec_total
- PR #1693²⁰⁶⁹ - Fix #1691: is_executor template specification fails for inherited executors
- PR #1692²⁰⁷⁰ - Fixed #1662: Possible exception source in coalescing_message_handler
- Issue #1691²⁰⁷¹ - is_executor template specification fails for inherited executors
- PR #1690²⁰⁷² - added macro for non-intrusive serialization of classes without a default c-tor
- PR #1689²⁰⁷³ - Replace value_or_error with custom storage, unify future_data state
- Issue #1688²⁰⁷⁴ - Add timer counters for tfunc_total and exec_total
- PR #1687²⁰⁷⁵ - Fixed interval timer
- PR #1686²⁰⁷⁶ - Fixing cmake warnings about not existing pseudo target dependencies
- PR #1685²⁰⁷⁷ - Converting partitioners to use bulk async execute
- PR #1683²⁰⁷⁸ - Adds a tool for inspect that checks for character limits
- PR #1682²⁰⁷⁹ - Change project name to (uppercase) HPX
- PR #1681²⁰⁸⁰ - Counter shortnames

²⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1704>

²⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1703>

²⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1702>

²⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1701>

²⁰⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1700>

²⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1699>

²⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1698>

²⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1697>

²⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1696>

²⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1695>

²⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1694>

²⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1693>

²⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1692>

²⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1691>

²⁰⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1690>

²⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1689>

²⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1688>

²⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1687>

²⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1686>

²⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1685>

²⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1683>

²⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1682>

²⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1681>

- PR #1680²⁰⁸¹ - Extended Non-intrusive Serialization to Ease Usage for Library Developers
- PR #1679²⁰⁸² - Working on 1544: More executor changes
- PR #1678²⁰⁸³ - Transpose fixes
- PR #1677²⁰⁸⁴ - Improve Boost compatibility check
- PR #1676²⁰⁸⁵ - 1d stencil fix
- Issue #1675²⁰⁸⁶ - hpx project name is not HPX
- PR #1674²⁰⁸⁷ - Fixing the MPI parcellport
- PR #1673²⁰⁸⁸ - added move semantics to map/vector deserialization
- PR #1672²⁰⁸⁹ - Vs2015 await
- PR #1671²⁰⁹⁰ - Adapt transform for #1668
- PR #1670²⁰⁹¹ - Started to work on #1668
- PR #1669²⁰⁹² - Add this_thread_executors
- Issue #1667²⁰⁹³ - Apple build instructions in docs are out of date
- PR #1666²⁰⁹⁴ - Apex integration
- PR #1665²⁰⁹⁵ - Fixes an error with the whitespace check that showed the incorrect location of the error
- Issue #1664²⁰⁹⁶ - Inspect tool found incorrect endline whitespace
- PR #1663²⁰⁹⁷ - Improve use of locks
- Issue #1662²⁰⁹⁸ - Possible exception source in coalescing_message_handler
- PR #1661²⁰⁹⁹ - Added support for 128bit number serialization
- PR #1660²¹⁰⁰ - Serialization 128bits
- PR #1659²¹⁰¹ - Implemented inner_product and adjacent_diff algos
- PR #1658²¹⁰² - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1657²¹⁰³ - Use of shared_ptr in io_service_pool changed to unique_ptr

²⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1680>

²⁰⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1679>

²⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1678>

²⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1677>

²⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1676>

²⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1675>

²⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1674>

²⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1673>

²⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1672>

²⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1671>

²⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1670>

²⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1669>

²⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1667>

²⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1666>

²⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1665>

²⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1664>

²⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1663>

²⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1662>

²⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1661>

²¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1660>

²¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1659>

²¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1658>

²¹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1657>

- Issue #1656²¹⁰⁴ - 1d_stencil codes all have wrong factor
- PR #1654²¹⁰⁵ - When using runtime_mode_connect, find the correct localhost public ip address
- PR #1653²¹⁰⁶ - Fixing 1617
- PR #1652²¹⁰⁷ - Remove traits::action_may_require_id_splitting
- PR #1651²¹⁰⁸ - Fixed performance counters related to AGAS cache timings
- PR #1650²¹⁰⁹ - Remove leftovers of traits::type_size
- PR #1649²¹¹⁰ - Shorten target names on Windows to shorten used path names
- PR #1648²¹¹¹ - Fixing problems introduced by merging #1623 for older compilers
- PR #1647²¹¹² - Simplify running automatic builds on Windows
- Issue #1646²¹¹³ - Cache insert and update performance counters are broken
- Issue #1644²¹¹⁴ - Remove leftovers of traits::type_size
- Issue #1643²¹¹⁵ - Remove traits::action_may_require_id_splitting
- PR #1642²¹¹⁶ - Adds spell checker to the inspect tool for qbk and doxygen comments
- PR #1640²¹¹⁷ - First step towards fixing 688
- PR #1639²¹¹⁸ - Re-apply remaining changes from limit_dataflow_recursion branch
- PR #1638²¹¹⁹ - This fixes possible deadlock in the test ignore_while_locked_1485
- PR #1637²¹²⁰ - Fixing hpx::wait_all() invoked with two vector<future<T>>
- PR #1636²¹²¹ - Partially re-apply changes from limit_dataflow_recursion branch
- PR #1635²¹²² - Adding missing test for #1572
- PR #1634²¹²³ - Revert “Limit recursion-depth in dataflow to a configurable constant”
- PR #1633²¹²⁴ - Add command line option to ignore batch environment
- PR #1631²¹²⁵ - hpx::lcos::queue exhibits strange behavior
- PR #1630²¹²⁶ - Fixed endlene_whitespace_check.cpp to detect lines with only whitespace

²¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1656>

²¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1654>

²¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1653>

²¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1652>

²¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1651>

²¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1650>

²¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1649>

²¹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1648>

²¹¹² <https://github.com/STELLAR-GROUP/hpx/pull/1647>

²¹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1646>

²¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1644>

²¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1643>

²¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1642>

²¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1640>

²¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1639>

²¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1638>

²¹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1637>

²¹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1636>

²¹²² <https://github.com/STELLAR-GROUP/hpx/pull/1635>

²¹²³ <https://github.com/STELLAR-GROUP/hpx/pull/1634>

²¹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1633>

²¹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1631>

²¹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1630>

- [Issue #1629](#)²¹²⁷ - Inspect trailing whitespace checker problem
- [PR #1628](#)²¹²⁸ - Removed meaningless const qualifiers. Minor icpc fix.
- [PR #1627](#)²¹²⁹ - Fixing the queue LCO and add example demonstrating its use
- [PR #1626](#)²¹³⁰ - Deprecating `get_gid()`, add `get_id()` and `get_unmanaged_id()`
- [PR #1625](#)²¹³¹ - Allowing to specify whether to send credits along with message
- [Issue #1624](#)²¹³² - Lifetime issue
- [Issue #1623](#)²¹³³ - `hpx::wait_all()` invoked with two `vector<future<T>>` fails
- [PR #1622](#)²¹³⁴ - Executor partitioners
- [PR #1621](#)²¹³⁵ - Clean up coroutines implementation
- [Issue #1620](#)²¹³⁶ - Revert #1535
- [PR #1619](#)²¹³⁷ - Fix result type calculation for `hpx::make_continuation`
- [PR #1618](#)²¹³⁸ - Fixing RDTSC on Xeon/Phi
- [Issue #1617](#)²¹³⁹ - `hpx cmake` not working when run as a subproject
- [Issue #1616](#)²¹⁴⁰ - `cmake` problem resulting in RDTSC not working correctly for Xeon Phi creates very strange results for duration counters
- [Issue #1615](#)²¹⁴¹ - `hpx::make_continuation` requires input and output to be the same
- [PR #1614](#)²¹⁴² - Fixed remove copy test
- [Issue #1613](#)²¹⁴³ - Dataflow causes stack overflow
- [PR #1612](#)²¹⁴⁴ - Modified foreach partitioner to use bulk execute
- [PR #1611](#)²¹⁴⁵ - Limit recursion-depth in dataflow to a configurable constant
- [PR #1610](#)²¹⁴⁶ - Increase timeout for CircleCI
- [PR #1609](#)²¹⁴⁷ - Refactoring thread manager, mainly extracting thread pool
- [PR #1608](#)²¹⁴⁸ - Fixed running multiple localities without localities parameter
- [PR #1607](#)²¹⁴⁹ - More algorithm fixes to `adjacentfind`

²¹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1629>

²¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1628>

²¹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1627>

²¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1626>

²¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1625>

²¹³² <https://github.com/STELLAR-GROUP/hpx/issues/1624>

²¹³³ <https://github.com/STELLAR-GROUP/hpx/issues/1623>

²¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1622>

²¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1621>

²¹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1620>

²¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1619>

²¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1618>

²¹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1617>

²¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1616>

²¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1615>

²¹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1614>

²¹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1613>

²¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1612>

²¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1611>

²¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1610>

²¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1609>

²¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1608>

²¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1607>

- [Issue #1606](#)²¹⁵⁰ - Running without localities parameter binds to bogus port range
- [Issue #1605](#)²¹⁵¹ - Too many serializations
- [PR #1604](#)²¹⁵² - Changes the HPX image into a hyperlink
- [PR #1601](#)²¹⁵³ - Fixing problems with remove_copy algorithm tests
- [PR #1600](#)²¹⁵⁴ - Actions with ids cleanup
- [PR #1599](#)²¹⁵⁵ - Duplicate binding of global ids should fail
- [PR #1598](#)²¹⁵⁶ - Fixing array access
- [PR #1597](#)²¹⁵⁷ - Improved the reliability of connecting/disconnecting localities
- [Issue #1596](#)²¹⁵⁸ - Duplicate id binding should fail
- [PR #1595](#)²¹⁵⁹ - Fixing more cmake config constants
- [PR #1594](#)²¹⁶⁰ - Fixing preprocessor constant used to enable C++11 chrono
- [PR #1593](#)²¹⁶¹ - Adding operator() for hpx::launch
- [Issue #1592](#)²¹⁶² - Error (typo) in the docs
- [Issue #1590](#)²¹⁶³ - CMake fails when CMAKE_BINARY_DIR contains '+'.
This issue was resolved by [PR #1588](#)²¹⁶⁵.
- [Issue #1589](#)²¹⁶⁴ - Disconnecting a locality results in segfault using heartbeat example
- [PR #1588](#)²¹⁶⁵ - Fix doc string for config option HPX_WITH_EXAMPLES
- [PR #1586](#)²¹⁶⁶ - Fixing 1493
- [PR #1585](#)²¹⁶⁷ - Additional Check for Inspect Tool to detect Endline Whitespace
- [Issue #1584](#)²¹⁶⁸ - Clean up coroutines implementation
- [PR #1583](#)²¹⁶⁹ - Adding a check for end line whitespace
- [PR #1582](#)²¹⁷⁰ - Attempt to fix assert firing after scheduling loop was exited
- [PR #1581](#)²¹⁷¹ - Fixed adjacentfind_binary test
- [PR #1580](#)²¹⁷² - Prevent some of the internal cmake lists from growing indefinitely

²¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1606>

²¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1605>

²¹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1604>

²¹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1601>

²¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1600>

²¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1599>

²¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1598>

²¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1597>

²¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1596>

²¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1595>

²¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1594>

²¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1593>

²¹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1592>

²¹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1590>

²¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1589>

²¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1588>

²¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1586>

²¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1585>

²¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1584>

²¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1583>

²¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1582>

²¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1581>

²¹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1580>

- [PR #1579](#)²¹⁷³ - Removing type_size trait, replacing it with special archive type
- [Issue #1578](#)²¹⁷⁴ - Remove demangle_helper
- [PR #1577](#)²¹⁷⁵ - Get ptr problems
- [Issue #1576](#)²¹⁷⁶ - Refactor async, dataflow, and future::then
- [PR #1575](#)²¹⁷⁷ - Fixing tests for parallel rotate
- [PR #1574](#)²¹⁷⁸ - Cleaning up schedulers
- [PR #1573](#)²¹⁷⁹ - Fixing thread pool executor
- [PR #1572](#)²¹⁸⁰ - Fixing number of configured localities
- [PR #1571](#)²¹⁸¹ - Reimplement decay
- [PR #1570](#)²¹⁸² - Refactoring async, apply, and dataflow APIs
- [PR #1569](#)²¹⁸³ - Changed range for mach-o library lookup
- [PR #1568](#)²¹⁸⁴ - Mark decltype support as required
- [PR #1567](#)²¹⁸⁵ - Removed const from algorithms
- [Issue #1566](#)²¹⁸⁶ - CMAKE Configuration Test Failures for clang 3.5 on debian
- [PR #1565](#)²¹⁸⁷ - Dylib support
- [PR #1564](#)²¹⁸⁸ - Converted partitioners and some algorithms to use executors
- [PR #1563](#)²¹⁸⁹ - Fix several #includes for Boost.Preprocessor
- [PR #1562](#)²¹⁹⁰ - Adding configuration option disabling/enabling all message handlers
- [PR #1561](#)²¹⁹¹ - Removed all occurrences of boost::move replacing it with std::move
- [Issue #1560](#)²¹⁹² - Leftover HPX_REGISTER_ACTION_DECLARATION_2
- [PR #1558](#)²¹⁹³ - Revisit async/apply SFINAE conditions
- [PR #1557](#)²¹⁹⁴ - Removing type_size trait, replacing it with special archive type
- [PR #1556](#)²¹⁹⁵ - Executor algorithms

²¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1579>

²¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1578>

²¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1577>

²¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1576>

²¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1575>

²¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1574>

²¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1573>

²¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1572>

²¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1571>

²¹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1570>

²¹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1569>

²¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1568>

²¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1567>

²¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1566>

²¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1565>

²¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1564>

²¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1563>

²¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1562>

²¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1561>

²¹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1560>

²¹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1558>

²¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1557>

²¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1556>

- [PR #1555](#)²¹⁹⁶ - Remove the necessity to specify archive flags on the receiving end
- [PR #1554](#)²¹⁹⁷ - Removing obsolete Boost.Serialization macros
- [PR #1553](#)²¹⁹⁸ - Properly fix HPX_DEFINE_*_ACTION macros
- [PR #1552](#)²¹⁹⁹ - Fixed algorithms relying on copy_if implementation
- [PR #1551](#)²²⁰⁰ - PxfS - Modifying FindOrangeFS.cmake based on OrangeFS 2.9.X
- [Issue #1550](#)²²⁰¹ - Passing plain identifier inside HPX_DEFINE_PLAIN_ACTION_1
- [PR #1549](#)²²⁰² - Fixing intel14/libstdc++4.4
- [PR #1548](#)²²⁰³ - Moving raw_ptr to detail namespace
- [PR #1547](#)²²⁰⁴ - Adding support for executors to future.then
- [PR #1546](#)²²⁰⁵ - Executor traits result types
- [PR #1545](#)²²⁰⁶ - Integrate executors with dataflow
- [PR #1543](#)²²⁰⁷ - Fix potential zero-copy for primarynamespace::bulk_service_async et.al.
- [PR #1542](#)²²⁰⁸ - Merging HPX0.9.10 into pxfs branch
- [PR #1541](#)²²⁰⁹ - Removed stale cmake tests, unused since the great cmake refactoring
- [PR #1540](#)²²¹⁰ - Fix idle-rate on platforms without TSC
- [PR #1539](#)²²¹¹ - Reporting situation if zero-copy-serialization was performed by a parcel generated from a plain apply/async
- [PR #1538](#)²²¹² - Changed return type of bulk executors and added test
- [Issue #1537](#)²²¹³ - Incorrect cpuid config tests
- [PR #1536](#)²²¹⁴ - Changed return type of bulk executors and added test
- [PR #1535](#)²²¹⁵ - Make sure promise::get_gid() can be called more than once
- [PR #1534](#)²²¹⁶ - Fixed async_callback with bound callback
- [PR #1533](#)²²¹⁷ - Updated the link in the documentation to a publically- accessible URL
- [PR #1532](#)²²¹⁸ - Make sure sync primitives are not copyable nor movable

²¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1555>
²¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1554>
²¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1553>
²¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1552>
²²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1551>
²²⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1550>
²²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1549>
²²⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1548>
²²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1547>
²²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1546>
²²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1545>
²²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1543>
²²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1542>
²²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1541>
²²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1540>
²²¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1539>
²²¹² <https://github.com/STELLAR-GROUP/hpx/pull/1538>
²²¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1537>
²²¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1536>
²²¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1535>
²²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1534>
²²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1533>
²²¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1532>

- PR #1531²²¹⁹ - Fix unwrapped issue with future ranges of void type
- PR #1530²²²⁰ - Serialization complex
- Issue #1528²²²¹ - Unwrapped issue with future<void>
- Issue #1527²²²² - HPX does not build with Boost 1.58.0
- PR #1526²²²³ - Added support for boost.multi_array serialization
- PR #1525²²²⁴ - Properly handle deferred futures, fixes #1506
- PR #1524²²²⁵ - Making sure invalid action argument types generate clear error message
- Issue #1522²²²⁶ - Need serialization support for boost multi array
- Issue #1521²²²⁷ - Remote async and zero-copy serialization optimizations don't play well together
- PR #1520²²²⁸ - Fixing UB while registering polymorphic classes for serialization
- PR #1519²²²⁹ - Making detail::condition_variable safe to use
- PR #1518²²³⁰ - Fix when_some bug missing indices in its result
- Issue #1517²²³¹ - Typo may affect CMake build system tests
- PR #1516²²³² - Fixing Posix context
- PR #1515²²³³ - Fixing Posix context
- PR #1514²²³⁴ - Correct problems with loading dynamic components
- PR #1513²²³⁵ - Fixing intel glibc4 4
- Issue #1508²²³⁶ - memory and papi counters do not work
- Issue #1507²²³⁷ - Unrecognized Command Line Option Error causing exit status 0
- Issue #1506²²³⁸ - Properly handle deferred futures
- PR #1505²²³⁹ - Adding #include - would not compile without this
- Issue #1502²²⁴⁰ - boost::filesystem::exists throws unexpected exception
- Issue #1501²²⁴¹ - hwloc configuration options are wrong for MIC

²²¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1531>

²²²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1530>

²²²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1528>

²²²² <https://github.com/STELLAR-GROUP/hpx/issues/1527>

²²²³ <https://github.com/STELLAR-GROUP/hpx/pull/1526>

²²²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1525>

²²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1524>

²²²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1522>

²²²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1521>

²²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1520>

²²²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1519>

²²³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1518>

²²³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1517>

²²³² <https://github.com/STELLAR-GROUP/hpx/pull/1516>

²²³³ <https://github.com/STELLAR-GROUP/hpx/pull/1515>

²²³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1514>

²²³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1513>

²²³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1508>

²²³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1507>

²²³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1506>

²²³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1505>

²²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1502>

²²⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1501>

- PR #1504²²⁴² - Making sure `boost::filesystem::exists()` does not throw
- PR #1500²²⁴³ - Exit application on `--hpx:version/-v` and `--hpx:info`
- PR #1498²²⁴⁴ - Extended task block
- PR #1497²²⁴⁵ - Unique ptr serialization
- PR #1496²²⁴⁶ - Unique ptr serialization (closed)
- PR #1495²²⁴⁷ - Switching `circuleci` build type to debug
- Issue #1494²²⁴⁸ - `--hpx:version/-v` does not exit after printing version information
- Issue #1493²²⁴⁹ - add an `hpx_` prefix to libraries and components to avoid name conflicts
- Issue #1492²²⁵⁰ - Define and ensure limitations for arguments to `async/apply`
- PR #1489²²⁵¹ - Enable idle rate counter on demand
- PR #1488²²⁵² - Made sure `detail::condition_variable` can be safely destroyed
- PR #1487²²⁵³ - Introduced default (main) template implementation for `ignore_while_checking`
- PR #1486²²⁵⁴ - Add HPX inspect tool
- Issue #1485²²⁵⁵ - `ignore_while_locked` doesn't support all Lockable types
- PR #1484²²⁵⁶ - Docker image generation
- PR #1483²²⁵⁷ - Move external endian library into HPX
- PR #1482²²⁵⁸ - Actions with integer type ids
- Issue #1481²²⁵⁹ - Sync primitives safe destruction
- Issue #1480²²⁶⁰ - Move external/boost/endian into `hpx/util`
- Issue #1478²²⁶¹ - Boost inspect violations
- PR #1479²²⁶² - Adds serialization for arrays; some futher/minor fixes
- PR #1477²²⁶³ - Fixing problems with the Intel compiler using a GCC 4.4 std library
- PR #1476²²⁶⁴ - Adding `hpx::lcos::latch` and `hpx::lcos::local::latch`

²²⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1504>

²²⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1500>

²²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1498>

²²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1497>

²²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1496>

²²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1495>

²²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1494>

²²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1493>

²²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1492>

²²⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1489>

²²⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1488>

²²⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1487>

²²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1486>

²²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1485>

²²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1484>

²²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1483>

²²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1482>

²²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1481>

²²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1480>

²²⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1478>

²²⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1479>

²²⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1477>

²²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1476>

- Issue #1475²²⁶⁵ - Boost inspect violations
- PR #1473²²⁶⁶ - Fixing action move tests
- Issue #1471²²⁶⁷ - Sync primitives should not be movable
- PR #1470²²⁶⁸ - Removing `hpx::util::polymorphic_factory`
- PR #1468²²⁶⁹ - Fixed container creation
- Issue #1467²²⁷⁰ - HPX application fail during finalization
- Issue #1466²²⁷¹ - HPX doesn't pick up Torque's nodefile on SuperMIC
- Issue #1464²²⁷² - HPX option for pre and post bootstrap performance counters
- PR #1463²²⁷³ - Replacing `async_collocated(id, ...)` with `async(collocated(id), ...)`
- PR #1462²²⁷⁴ - Consolidated `task_region` with N4411
- PR #1461²²⁷⁵ - Consolidate inconsistent CMake option names
- Issue #1460²²⁷⁶ - Which `malloc` is actually used? or at least which one is HPX built with
- Issue #1459²²⁷⁷ - Make `cmake` configure step fail explicitly if compiler version is not supported
- Issue #1458²²⁷⁸ - Update `parallel::task_region` with N4411
- PR #1456²²⁷⁹ - Consolidating `new_<>()`
- Issue #1455²²⁸⁰ - Replace `async_collocated(id, ...)` with `async(collocated(id), ...)`
- PR #1454²²⁸¹ - Removed harmful `std::moves` from return statements
- PR #1453²²⁸² - Use range-based for-loop instead of `Boost.Foreach`
- PR #1452²²⁸³ - C++ feature tests
- PR #1451²²⁸⁴ - When serializing, pass archive flags to `traits::get_type_size`
- Issue #1450²²⁸⁵ - `traits::get_type_size` needs archive flags to enable `zero_copy` optimizations
- Issue #1449²²⁸⁶ - "couldn't create performance counter" - AGAS
- Issue #1448²²⁸⁷ - Replace distributing factories with `new_<T[]>(...)`

²²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1475>

²²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1473>

²²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1471>

²²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1470>

²²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1468>

²²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1467>

²²⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1466>

²²⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1464>

²²⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1463>

²²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1462>

²²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1461>

²²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1460>

²²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1459>

²²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1458>

²²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1456>

²²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1455>

²²⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1454>

²²⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1453>

²²⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1452>

²²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1451>

²²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1450>

²²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1449>

²²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1448>

- PR #1447²²⁸⁸ - Removing obsolete remote_object component
- PR #1446²²⁸⁹ - Hpx serialization
- PR #1445²²⁹⁰ - Replacing travis with circleci
- PR #1443²²⁹¹ - Always stripping HPX command line arguments before executing start function
- PR #1442²²⁹² - Adding `-hpx:bind=none` to disable thread affinities
- Issue #1439²²⁹³ - Libraries get linked in multiple times, RPATH is not properly set
- PR #1438²²⁹⁴ - Removed superfluous typedefs
- Issue #1437²²⁹⁵ - `hpx::init()` should strip HPX-related flags from argv
- Issue #1436²²⁹⁶ - Add strong scaling option to https
- PR #1435²²⁹⁷ - Adding `async_cb`, `async_continue_cb`, and `async_collocated_cb`
- PR #1434²²⁹⁸ - Added missing install rule, removed some dead CMake code
- PR #1433²²⁹⁹ - Add GitExternal and SubProject cmake scripts from eyescale/cmake repo
- Issue #1432²³⁰⁰ - Add command line flag to disable thread pinning
- PR #1431²³⁰¹ - Fix #1423
- Issue #1430²³⁰² - Inconsistent CMake option names
- Issue #1429²³⁰³ - Configure setting `HPX_HAVE_PARCELPORTR_MPI` is ignored
- PR #1428²³⁰⁴ - Fixes #1419 (closed)
- PR #1427²³⁰⁵ - Adding `stencil_iterator` and `transform_iterator`
- PR #1426²³⁰⁶ - Fixes #1419
- PR #1425²³⁰⁷ - During serialization memory allocation should honour allocator chunk size
- Issue #1424²³⁰⁸ - chunk allocation during serialization does not use memory pool/allocator chunk size
- Issue #1423²³⁰⁹ - Remove `HPX_STD_UNIQUE_PTR`
- Issue #1422²³¹⁰ - `hpx:threads=all` allocates too many os threads

²²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1447>

²²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1446>

²²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1445>

²²⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1443>

²²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1442>

²²⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1439>

²²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1438>

²²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1437>

²²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1436>

²²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1435>

²²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1434>

²²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1433>

²³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1432>

²³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1431>

²³⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1430>

²³⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1429>

²³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1428>

²³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1427>

²³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1426>

²³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1425>

²³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1424>

²³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1423>

²³¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1422>

- PR #1420²³¹¹ - added .travis.yml
- Issue #1419²³¹² - Unify enums: `hpx::runtime::state` and `hpx::state`
- PR #1416²³¹³ - Adding travis builder
- Issue #1414²³¹⁴ - Correct directory for `dispatch_gcc46.hpp` iteration
- Issue #1410²³¹⁵ - Set operation algorithms
- Issue #1389²³¹⁶ - Parallel algorithms relying on scan partitioner break for small number of elements
- Issue #1325²³¹⁷ - Exceptions thrown during parcel handling are not handled correctly
- Issue #1315²³¹⁸ - Errors while running performance tests
- Issue #1309²³¹⁹ - `hpx::vector` partitions are not easily extendable by applications
- PR #1300²³²⁰ - Added serialization/de-serialization to `examples.tuplespace`
- Issue #1251²³²¹ - `hpx::threads::get_thread_count` doesn't consider pending threads
- Issue #1008²³²² - Decrease in application performance overtime; occasional spikes of major slowdown
- Issue #1001²³²³ - Zero copy serialization raises assert
- Issue #721²³²⁴ - Make HPX usable for Xeon Phi
- Issue #524²³²⁵ - 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.

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

²³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1420>

²³¹² <https://github.com/STELLAR-GROUP/hpx/issues/1419>

²³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1416>

²³¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1414>

²³¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1410>

²³¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1389>

²³¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1325>

²³¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1315>

²³¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1309>

²³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1300>

²³²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1251>

²³²² <https://github.com/STELLAR-GROUP/hpx/issues/1008>

²³²³ <https://github.com/STELLAR-GROUP/hpx/issues/1001>

²³²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/721>

²³²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/524>

and to improve the available networking bandwidth. Please note that in this release we disabled the `ibverbs` and `ipc` parcel ports as those have not been ported to the new plugin system yet (see [Issue #839](#)²³²⁶).

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](#)²³²⁷ 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_nonsr` which was the default expansion used for this macro. All `HPX` API functions which expect a `hpx::util::function_nonsr` (or a `hpx::util::unique_function_nonsr`) can now be transparently called with a compatible `std::function` instead. Similarly, `HPX_STD_TUPLE` was replaced by its default expansion as well: `hpx::util::tuple`.
- Changes to `hpx::unique_future`- `hpx::unique_future`, which was deprecated in the previous release for `hpx::future` is now completely removed from `HPX`. This completes the transition to a completely standards conforming implementation of `hpx::future`.
- Changes to Supported Compilers. Finally, in order to utilize more C++11 semantics, we have officially dropped support for GCC 4.4 and MSVC 2012. Please see our [Prerequisites](#) page for more details.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- [Issue #1402](#)²³²⁸ - Internal shared_future serialization copies
- [Issue #1399](#)²³²⁹ - Build takes unusually long time...
- [Issue #1398](#)²³³⁰ - Tests using the scan partitioner are broken on at least gcc 4.7 and intel compiler
- [Issue #1397](#)²³³¹ - Completely remove `hpx::unique_future`
- [Issue #1396](#)²³³² - Parallel scan algorithms with different initial values

²³²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/839>

²³²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1338>

²³²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1402>

²³²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1399>

²³³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1398>

²³³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1397>

²³³² <https://github.com/STELLAR-GROUP/hpx/issues/1396>

- Issue #1395²³³³ - Race Condition - 1d_stencil_8 - SuperMIC
- Issue #1394²³³⁴ - “suspending thread while at least one lock is being held” - 1d_stencil_8 - SuperMIC
- Issue #1393²³³⁵ - SEGFAULT in 1d_stencil_8 on SuperMIC
- Issue #1392²³³⁶ - Fixing #1168
- Issue #1391²³³⁷ - Parallel Algorithms for scan partitioner for small number of elements
- Issue #1387²³³⁸ - Failure with more than 4 localities
- Issue #1386²³³⁹ - Dispatching unhandled exceptions to outer user code
- Issue #1385²³⁴⁰ - Adding Copy algorithms, fixing `parallel::copy_if`
- Issue #1384²³⁴¹ - Fixing 1325
- Issue #1383²³⁴² - Fixed #504: Refactor Dataflow LCO to work with futures, this removes the dataflow component as it is obsolete
- Issue #1382²³⁴³ - `is_sorted`, `is_sorted_until` and `is_partitioned` algorithms
- Issue #1381²³⁴⁴ - fix for CMake versions prior to 3.1
- Issue #1380²³⁴⁵ - resolved warning in CMake 3.1 and newer
- Issue #1379²³⁴⁶ - Compilation error with papi
- Issue #1378²³⁴⁷ - Towards safer migration
- Issue #1377²³⁴⁸ - HPXConfig.cmake should include `TCMALLOC_LIBRARY` and `TCMALLOC_INCLUDE_DIR`
- Issue #1376²³⁴⁹ - Warning on uninitialized member
- Issue #1375²³⁵⁰ - Fixing 1163
- Issue #1374²³⁵¹ - Fixing the MSVC 12 release builder
- Issue #1373²³⁵² - Modifying parallel search algorithm for zero length searches
- Issue #1372²³⁵³ - Modifying parallel search algorithm for zero length searches
- Issue #1371²³⁵⁴ - Avoid holding a lock during `agas::incf` while doing a credit split
- Issue #1370²³⁵⁵ - `--hpx:bind` throws unexpected error

²³³³ <https://github.com/STELLAR-GROUP/hpx/issues/1395>

²³³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1394>

²³³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1393>

²³³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1392>

²³³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1391>

²³³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1387>

²³³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1386>

²³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1385>

²³⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1384>

²³⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1383>

²³⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1382>

²³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1381>

²³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1380>

²³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1379>

²³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1378>

²³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1377>

²³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1376>

²³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1375>

²³⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1374>

²³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1373>

²³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1372>

²³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1371>

²³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1370>

- Issue #1369²³⁵⁶ - Getting rid of (void) in loops
- Issue #1368²³⁵⁷ - Variadic templates support for tuple
- Issue #1367²³⁵⁸ - One last batch of variadic templates support
- Issue #1366²³⁵⁹ - Fixing symbolic namespace hang
- Issue #1365²³⁶⁰ - More held locks
- Issue #1364²³⁶¹ - Add counters 1363
- Issue #1363²³⁶² - Add thread overhead counters
- Issue #1362²³⁶³ - Std config removal
- Issue #1361²³⁶⁴ - Parcelport plugins
- Issue #1360²³⁶⁵ - Detuplify transfer_action
- Issue #1359²³⁶⁶ - Removed obsolete checks
- Issue #1358²³⁶⁷ - Fixing 1352
- Issue #1357²³⁶⁸ - Variadic templates support for runtime_support and components
- Issue #1356²³⁶⁹ - fixed coordinate test for intel13
- Issue #1355²³⁷⁰ - fixed coordinate.hpp
- Issue #1354²³⁷¹ - Lexicographical Compare completed
- Issue #1353²³⁷² - HPX should set Boost_ADDITIONAL_VERSIONS flags
- Issue #1352²³⁷³ - Error: Cannot find action “” in type registry: HPX(bad_action_code)
- Issue #1351²³⁷⁴ - Variadic templates support for appliers
- Issue #1350²³⁷⁵ - Actions simplification
- Issue #1349²³⁷⁶ - Variadic when and wait functions
- Issue #1348²³⁷⁷ - Added hpx_init header to test files
- Issue #1347²³⁷⁸ - Another batch of variadic templates support

²³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1369>

²³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1368>

²³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1367>

²³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1366>

²³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1365>

²³⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1364>

²³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1363>

²³⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1362>

²³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1361>

²³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1360>

²³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1359>

²³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1358>

²³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1357>

²³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1356>

²³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1355>

²³⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1354>

²³⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1353>

²³⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1352>

²³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1351>

²³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1350>

²³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1349>

²³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1348>

²³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1347>

- Issue #1346²³⁷⁹ - Segmented copy
- Issue #1345²³⁸⁰ - Attempting to fix hangs during shutdown
- Issue #1344²³⁸¹ - Std config removal
- Issue #1343²³⁸² - Removing various distribution policies for `hpx::vector`
- Issue #1342²³⁸³ - Inclusive scan
- Issue #1341²³⁸⁴ - Exclusive scan
- Issue #1340²³⁸⁵ - Adding `parallel::count` for distributed data structures, adding tests
- Issue #1339²³⁸⁶ - Update argument order for `transform_reduce`
- Issue #1337²³⁸⁷ - Fix dataflow to handle properly ranges of futures
- Issue #1336²³⁸⁸ - dataflow needs to hold onto futures passed to it
- Issue #1335²³⁸⁹ - Fails to compile with `msvc14`
- Issue #1334²³⁹⁰ - Examples build problem
- Issue #1333²³⁹¹ - Distributed transform reduce
- Issue #1332²³⁹² - Variadic templates support for actions
- Issue #1331²³⁹³ - Some ambiguous calls of `map::erase` have been prevented by adding additional check in locality constructor.
- Issue #1330²³⁹⁴ - Defining Plain Actions does not work as described in the documentation
- Issue #1329²³⁹⁵ - Distributed vector cleanup
- Issue #1328²³⁹⁶ - Sync docs and comments with code in `hello_world` example
- Issue #1327²³⁹⁷ - Typos in docs
- Issue #1326²³⁹⁸ - Documentation and code diverged in Fibonacci tutorial
- Issue #1325²³⁹⁹ - Exceptions thrown during parcel handling are not handled correctly
- Issue #1324²⁴⁰⁰ - fixed bandwidth calculation
- Issue #1323²⁴⁰¹ - `mmap()` failed to allocate thread stack due to insufficient resources

²³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1346>

²³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1345>

²³⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1344>

²³⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1343>

²³⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1342>

²³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1341>

²³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1340>

²³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1339>

²³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1337>

²³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1336>

²³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1335>

²³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1334>

²³⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1333>

²³⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1332>

²³⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1331>

²³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1330>

²³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1329>

²³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1328>

²³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1327>

²³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1326>

²³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1325>

²⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1324>

²⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1323>

- [Issue #1322](#)²⁴⁰² - HPX fails to build aa182cf
- [Issue #1321](#)²⁴⁰³ - Limiting size of outgoing messages while coalescing parcels
- [Issue #1320](#)²⁴⁰⁴ - passing a future with `launch::deferred` in remote function call causes hang
- [Issue #1319](#)²⁴⁰⁵ - An exception when tries to specify number high priority threads with `abp-priority`
- [Issue #1318](#)²⁴⁰⁶ - Unable to run program with `abp-priority` and `numa-sensitivity` enabled
- [Issue #1317](#)²⁴⁰⁷ - N4071 `Search/Search_n` finished, minor changes
- [Issue #1316](#)²⁴⁰⁸ - Add config option to make `-lhp.run_hpx_main!=1` the default
- [Issue #1314](#)²⁴⁰⁹ - Variadic support for `async` and `apply`
- [Issue #1313](#)²⁴¹⁰ - Adjust `when_any/some` to the latest proposed interfaces
- [Issue #1312](#)²⁴¹¹ - Fixing #857: `hpx::naming::locality` leaks `parcelport` specific information into the public interface
- [Issue #1311](#)²⁴¹² - Distributed `get'er/set'er_values` for distributed vector
- [Issue #1310](#)²⁴¹³ - Crashing in `hpx::parcelset::policies::mpi::connection_handler::handle_messages()` on Super-MIC
- [Issue #1308](#)²⁴¹⁴ - Unable to execute an application with `-hpx:threads`
- [Issue #1307](#)²⁴¹⁵ - `merge_graph` linking issue
- [Issue #1306](#)²⁴¹⁶ - First batch of variadic templates support
- [Issue #1305](#)²⁴¹⁷ - Create a compiler wrapper
- [Issue #1304](#)²⁴¹⁸ - Provide a compiler wrapper for `hpx`
- [Issue #1303](#)²⁴¹⁹ - Drop support for GCC44
- [Issue #1302](#)²⁴²⁰ - Fixing #1297
- [Issue #1301](#)²⁴²¹ - Compilation error when tried to use `boost range` iterators with `wait_all`
- [Issue #1298](#)²⁴²² - Distributed vector
- [Issue #1297](#)²⁴²³ - Unable to invoke component actions recursively

²⁴⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1322>

²⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1321>

²⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1320>

²⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1319>

²⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1318>

²⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1317>

²⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1316>

²⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1314>

²⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1313>

²⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1312>

²⁴¹² <https://github.com/STELLAR-GROUP/hpx/issues/1311>

²⁴¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1310>

²⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1308>

²⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1307>

²⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1306>

²⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1305>

²⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1304>

²⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1303>

²⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1302>

²⁴²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1301>

²⁴²² <https://github.com/STELLAR-GROUP/hpx/issues/1298>

²⁴²³ <https://github.com/STELLAR-GROUP/hpx/issues/1297>

- Issue #1294²⁴²⁴ - HDF5 build error
- Issue #1275²⁴²⁵ - The parcelport implementation is non-optimal
- Issue #1267²⁴²⁶ - Added classes and unit tests for local_file, orangefs_file and pxfs_file
- Issue #1264²⁴²⁷ - Error “assertion ‘!m_fun’ failed” randomly occurs when using TCP
- Issue #1254²⁴²⁸ - thread binding seems to not work properly
- Issue #1220²⁴²⁹ - parallel::copy_if is broken
- Issue #1217²⁴³⁰ - Find a better way of fixing the issue patched by #1216
- Issue #1168²⁴³¹ - Starting HPX on Cray machines using aprun isn’t working correctly
- Issue #1085²⁴³² - Replace startup and shutdown barriers with broadcasts
- Issue #981²⁴³³ - With SLURM, -hpx:threads=8 should not be necessary
- Issue #857²⁴³⁴ - hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #850²⁴³⁵ - “flush” not documented
- Issue #763²⁴³⁶ - Create buildbot instance that uses std::bind as HPX_STD_BIND
- Issue #680²⁴³⁷ - Convert parcel ports into a plugin system
- Issue #582²⁴³⁸ - Make exception thrown from HPX threads available from hpx::init
- Issue #504²⁴³⁹ - Refactor Dataflow LCO to work with futures
- Issue #196²⁴⁴⁰ - Don’t store copies of the locality network metadata in the gva table

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.

²⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1294>

²⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1275>

²⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1267>

²⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1264>

²⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1254>

²⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1220>

²⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1217>

²⁴³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1168>

²⁴³² <https://github.com/STELLAR-GROUP/hpx/issues/1085>

²⁴³³ <https://github.com/STELLAR-GROUP/hpx/issues/981>

²⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/857>

²⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/850>

²⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/763>

²⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/680>

²⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/582>

²⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/504>

²⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/196>

- We rewrote and significantly changed our build system. Please have a look at the new (now generated) documentation here: [HPX build system](#). Please revisit your build scripts to adapt to the changes. The most notable changes are:
 - HPX_NO_INSTALL is no longer necessary.
 - For external builds, you need to set HPX_DIR instead of HPX_ROOT as described here: [Using HPX with CMake-based projects](#).
 - IDEs that support multiple configurations (Visual Studio and XCode) can now be used as intended. that means no build dir.
 - Building HPX statically (without dynamic libraries) is now supported (-DHPX_STATIC_LINKING=On).
 - Please note that many variables used to configure the build process have been renamed to unify the naming conventions (see the section [CMake variables used to configure HPX](#) for more information).
 - This also fixes a long list of issues, for more information see [Issue #1204](#)²⁴⁴¹.
- We started to implement various proposals to the C++ Standardization committee related to parallelism and concurrency, most notably [N4409](#)²⁴⁴² (Working Draft, Technical Specification for C++ Extensions for Parallelism), [N4411](#)²⁴⁴³ (Task Region Rev. 3), and [N4313](#)²⁴⁴⁴ (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](#)²⁴⁴⁵.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- [Issue #1296](#)²⁴⁴⁶ - Rename make_error_future to make_exceptional_future, adjust to N4123
- [Issue #1295](#)²⁴⁴⁷ - building issue
- [Issue #1293](#)²⁴⁴⁸ - Transpose example
- [Issue #1292](#)²⁴⁴⁹ - Wrong abs() function used in example
- [Issue #1291](#)²⁴⁵⁰ - non-synchronized shift operators have been removed
- [Issue #1290](#)²⁴⁵¹ - RDTSCP is defined as true for Xeon Phi build
- [Issue #1289](#)²⁴⁵² - Fixing 1288
- [Issue #1288](#)²⁴⁵³ - Add new performance counters
- [Issue #1287](#)²⁴⁵⁴ - Hierarchy scheduler broken performance counters

²⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1204>

²⁴⁴² <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf>

²⁴⁴³ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

²⁴⁴⁴ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

²⁴⁴⁵ <https://roostam.cct.lsu.edu/>

²⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1296>

²⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1295>

²⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1293>

²⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1292>

²⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1291>

²⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1290>

²⁴⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1289>

²⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1288>

²⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1287>

- Issue #1286²⁴⁵⁵ - Algorithm cleanup
- Issue #1285²⁴⁵⁶ - Broken Links in Documentation
- Issue #1284²⁴⁵⁷ - Uninitialized copy
- Issue #1283²⁴⁵⁸ - missing boost::scoped_ptr includes
- Issue #1282²⁴⁵⁹ - Update documentation of build options for schedulers
- Issue #1281²⁴⁶⁰ - reset idle rate counter
- Issue #1280²⁴⁶¹ - Bug when executing on Intel MIC
- Issue #1279²⁴⁶² - Add improved when_all/wait_all
- Issue #1278²⁴⁶³ - Implement improved when_all/wait_all
- Issue #1277²⁴⁶⁴ - feature request: get access to argc argv and variables_map
- Issue #1276²⁴⁶⁵ - Remove merging map
- Issue #1274²⁴⁶⁶ - Weird (wrong) string code in papi.cpp
- Issue #1273²⁴⁶⁷ - Sequential task execution policy
- Issue #1272²⁴⁶⁸ - Avoid CMake name clash for Boost.Thread library
- Issue #1271²⁴⁶⁹ - Updates on HPX Test Units
- Issue #1270²⁴⁷⁰ - hpx/util/safe_lexical_cast.hpp is added
- Issue #1269²⁴⁷¹ - Added default value for “LIB” cmake variable
- Issue #1268²⁴⁷² - Memory Counters not working
- Issue #1266²⁴⁷³ - FindHPX.cmake is not installed
- Issue #1263²⁴⁷⁴ - apply_remote test takes too long
- Issue #1262²⁴⁷⁵ - Chrono cleanup
- Issue #1261²⁴⁷⁶ - Need make install for papi counters and this builds all the examples
- Issue #1260²⁴⁷⁷ - Documentation of Stencil example claims

²⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1286>

²⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1285>

²⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1284>

²⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1283>

²⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1282>

²⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1281>

²⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1280>

²⁴⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1279>

²⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1278>

²⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1277>

²⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1276>

²⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1274>

²⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1273>

²⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1272>

²⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1271>

²⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1270>

²⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1269>

²⁴⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1268>

²⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1266>

²⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1263>

²⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1262>

²⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1261>

²⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1260>

- [Issue #1259²⁴⁷⁸](#) - Avoid double-linking Boost on Windows
- [Issue #1257²⁴⁷⁹](#) - Adding additional parameter to create_thread
- [Issue #1256²⁴⁸⁰](#) - added buildbot changes to release notes
- [Issue #1255²⁴⁸¹](#) - Cannot build MiniGhost
- [Issue #1253²⁴⁸²](#) - hpx::thread defects
- [Issue #1252²⁴⁸³](#) - HPX_PREFIX is too fragile
- [Issue #1250²⁴⁸⁴](#) - switch_to_fiber_emulation does not work properly
- [Issue #1249²⁴⁸⁵](#) - Documentation is generated under Release folder
- [Issue #1248²⁴⁸⁶](#) - Fix usage of hpx_generic_coroutine_context and get tests passing on powerpc
- [Issue #1247²⁴⁸⁷](#) - Dynamic linking error
- [Issue #1246²⁴⁸⁸](#) - Make cpuid.cpp C++11 compliant
- [Issue #1245²⁴⁸⁹](#) - HPX fails on startup (setting thread affinity mask)
- [Issue #1244²⁴⁹⁰](#) - HPX_WITH_RDTSC configure test fails, but should succeed
- [Issue #1243²⁴⁹¹](#) - CTest dashboard info for CSCS CDash drop location
- [Issue #1242²⁴⁹²](#) - Mac fixes
- [Issue #1241²⁴⁹³](#) - Failure in Distributed with Boost 1.56
- [Issue #1240²⁴⁹⁴](#) - fix a race condition in examples.diskperf
- [Issue #1239²⁴⁹⁵](#) - fix wait_each in examples.diskperf
- [Issue #1238²⁴⁹⁶](#) - Fixed #1237: hpx::util::portable_binary_iarchive failed
- [Issue #1237²⁴⁹⁷](#) - hpx::util::portable_binary_iarchive faileds
- [Issue #1235²⁴⁹⁸](#) - Fixing clang warnings and errors
- [Issue #1234²⁴⁹⁹](#) - TCP runs fail: Transport endpoint is not connected
- [Issue #1233²⁵⁰⁰](#) - Making sure the correct number of threads is registered with AGAS

²⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1259>

²⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1257>

²⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1256>

²⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1255>

²⁴⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1253>

²⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1252>

²⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1250>

²⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1249>

²⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1248>

²⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1247>

²⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1246>

²⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1245>

²⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1244>

²⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1243>

²⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1242>

²⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1241>

²⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1240>

²⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1239>

²⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1238>

²⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1237>

²⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1235>

²⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1234>

²⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1233>

- Issue #1232²⁵⁰¹ - Fixing race in wait_xxx
- Issue #1231²⁵⁰² - Parallel minmax
- Issue #1230²⁵⁰³ - Distributed run of 1d_stencil_8 uses less threads than spec. & sometimes gives errors
- Issue #1229²⁵⁰⁴ - Unstable number of threads
- Issue #1228²⁵⁰⁵ - HPX link error (cmake / MPI)
- Issue #1226²⁵⁰⁶ - Warning about struct/class thread_counters
- Issue #1225²⁵⁰⁷ - Adding parallel::replace etc
- Issue #1224²⁵⁰⁸ - Extending dataflow to pass through non-future arguments
- Issue #1223²⁵⁰⁹ - Remaining find algorithms implemented, N4071
- Issue #1222²⁵¹⁰ - Merging all the changes
- Issue #1221²⁵¹¹ - No error output when using mpirun with hpx
- Issue #1219²⁵¹² - Adding new AGAS cache performance counters
- Issue #1216²⁵¹³ - Fixing using futures (clients) as arguments to actions
- Issue #1215²⁵¹⁴ - Error compiling simple component
- Issue #1214²⁵¹⁵ - Stencil docs
- Issue #1213²⁵¹⁶ - Using more than a few dozen MPI processes on SuperMike results in a seg fault before getting to hpx_main
- Issue #1212²⁵¹⁷ - Parallel rotate
- Issue #1211²⁵¹⁸ - Direct actions cause the future's shared_state to be leaked
- Issue #1210²⁵¹⁹ - Refactored local::promise to be standard conformant
- Issue #1209²⁵²⁰ - Improve command line handling
- Issue #1208²⁵²¹ - Adding parallel::reverse and parallel::reverse_copy
- Issue #1207²⁵²² - Add copy_backward and move_backward
- Issue #1206²⁵²³ - N4071 additional algorithms implemented

²⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1232>

²⁵⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1231>

²⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1230>

²⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1229>

²⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1228>

²⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1226>

²⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1225>

²⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1224>

²⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1223>

²⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1222>

²⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1221>

²⁵¹² <https://github.com/STELLAR-GROUP/hpx/issues/1219>

²⁵¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1216>

²⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1215>

²⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1214>

²⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1213>

²⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1212>

²⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1211>

²⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1210>

²⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1209>

²⁵²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1208>

²⁵²² <https://github.com/STELLAR-GROUP/hpx/issues/1207>

²⁵²³ <https://github.com/STELLAR-GROUP/hpx/issues/1206>

- [Issue #1204](#)²⁵²⁴ - Cmake simplification and various other minor changes
- [Issue #1203](#)²⁵²⁵ - Implementing new launch policy for (local) `async: hpx::launch::fork`.
- [Issue #1202](#)²⁵²⁶ - Failed assertion in `connection_cache.hpp`
- [Issue #1201](#)²⁵²⁷ - `pkg-config` doesn't add `mpi` link directories
- [Issue #1200](#)²⁵²⁸ - Error when querying time performance counters
- [Issue #1199](#)²⁵²⁹ - library path is now configurable (again)
- [Issue #1198](#)²⁵³⁰ - Error when querying performance counters
- [Issue #1197](#)²⁵³¹ - tests fail with intel compiler
- [Issue #1196](#)²⁵³² - Silence several warnings
- [Issue #1195](#)²⁵³³ - Rephrase initializers to work with VC++ 2012
- [Issue #1194](#)²⁵³⁴ - Simplify parallel algorithms
- [Issue #1193](#)²⁵³⁵ - Adding `parallel::equal`
- [Issue #1192](#)²⁵³⁶ - `HPX(out_of_memory)` on including `<hpx/hpx.hpp>`
- [Issue #1191](#)²⁵³⁷ - Fixing #1189
- [Issue #1190](#)²⁵³⁸ - Chrono cleanup
- [Issue #1189](#)²⁵³⁹ - Deadlock .. somewhere? (probably serialization)
- [Issue #1188](#)²⁵⁴⁰ - Removed `future::get_status()`
- [Issue #1186](#)²⁵⁴¹ - Fixed `FindOpenCL` to find current AMD APP SDK
- [Issue #1184](#)²⁵⁴² - Tweaking future unwrapping
- [Issue #1183](#)²⁵⁴³ - Extended `parallel::reduce`
- [Issue #1182](#)²⁵⁴⁴ - `future::unwrap` hangs for `launch::deferred`
- [Issue #1181](#)²⁵⁴⁵ - Adding `all_of`, `any_of`, and `none_of` and corresponding documentation
- [Issue #1180](#)²⁵⁴⁶ - `hpx::cout` defect

²⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1204>

²⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1203>

²⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1202>

²⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1201>

²⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1200>

²⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1199>

²⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1198>

²⁵³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1197>

²⁵³² <https://github.com/STELLAR-GROUP/hpx/issues/1196>

²⁵³³ <https://github.com/STELLAR-GROUP/hpx/issues/1195>

²⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1194>

²⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1193>

²⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1192>

²⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1191>

²⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1190>

²⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1189>

²⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1188>

²⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1186>

²⁵⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1184>

²⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1183>

²⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1182>

²⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1181>

²⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1180>

- Issue #1179²⁵⁴⁷ - `hpx::async` does not work for member function pointers when called on types with self-defined unary operator*
- Issue #1178²⁵⁴⁸ - Implemented variadic `hpx::util::zip_iterator`
- Issue #1177²⁵⁴⁹ - MPI parcellport defect
- Issue #1176²⁵⁵⁰ - `HPX_DEFINE_COMPONENT_CONST_ACTION_TPL` does not have a 2-argument version
- Issue #1175²⁵⁵¹ - Create `util::zip_iterator` working with `util::tuple<>`
- Issue #1174²⁵⁵² - Error Building HPX on linux, `root_certificate_authority.cpp`
- Issue #1173²⁵⁵³ - `hpx::cout` output lost
- Issue #1172²⁵⁵⁴ - HPX build error with Clang 3.4.2
- Issue #1171²⁵⁵⁵ - `CMAKE_INSTALL_PREFIX` ignored
- Issue #1170²⁵⁵⁶ - Close `hpx_benchmarks` repository on Github
- Issue #1169²⁵⁵⁷ - Buildbot emails have syntax error in url
- Issue #1167²⁵⁵⁸ - Merge partial implementation of standards proposal N3960
- Issue #1166²⁵⁵⁹ - Fixed several compiler warnings
- Issue #1165²⁵⁶⁰ - `cmake` warns: “`tests.regressions.actions`” does not exist
- Issue #1164²⁵⁶¹ - Want my own serialization of `hpx::future`
- Issue #1162²⁵⁶² - Segfault in `hello_world` example
- Issue #1161²⁵⁶³ - Use `HPX_ASSERT` to aid the compiler
- Issue #1160²⁵⁶⁴ - Do not put `-DNDEBUG` into `hpx_application.pc`
- Issue #1159²⁵⁶⁵ - Support Clang 3.4.2
- Issue #1158²⁵⁶⁶ - Fixed #1157: Rename `when_n/wait_n`, add `when_xxx_n/wait_xxx_n`
- Issue #1157²⁵⁶⁷ - Rename `when_n/wait_n`, add `when_xxx_n/wait_xxx_n`
- Issue #1156²⁵⁶⁸ - Force inlining fails
- Issue #1155²⁵⁶⁹ - changed header of `printout` to be compatible with python csv module

²⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1179>

²⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1178>

²⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1177>

²⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1176>

²⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1175>

²⁵⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1174>

²⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1173>

²⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1172>

²⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1171>

²⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1170>

²⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1169>

²⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1167>

²⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1166>

²⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1165>

²⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1164>

²⁵⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1162>

²⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1161>

²⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1160>

²⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1159>

²⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1158>

²⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1157>

²⁵⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1156>

²⁵⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1155>

- Issue #1154²⁵⁷⁰ - Fixing iostreams
- Issue #1153²⁵⁷¹ - Standard manipulators (like `std::endl`) do not work with `hpx::ostream`
- Issue #1152²⁵⁷² - Functions revamp
- Issue #1151²⁵⁷³ - Suppressing cmake 3.0 policy warning for CMP0026
- Issue #1150²⁵⁷⁴ - Client Serialization error
- Issue #1149²⁵⁷⁵ - Segfault on Stampede
- Issue #1148²⁵⁷⁶ - Refactoring mini-ghost
- Issue #1147²⁵⁷⁷ - N3960 `copy_if` and `copy_n` implemented and tested
- Issue #1146²⁵⁷⁸ - Stencil print
- Issue #1145²⁵⁷⁹ - N3960 `hpx::parallel::copy` implemented and tested
- Issue #1144²⁵⁸⁰ - OpenMP examples `1d_stencil` do not build
- Issue #1143²⁵⁸¹ - `1d_stencil` OpenMP examples do not build
- Issue #1142²⁵⁸² - Cannot build HPX with gcc 4.6 on OS X
- Issue #1140²⁵⁸³ - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
- Issue #1139²⁵⁸⁴ - `hpx/hpx/config/compiler_specific.hpp`
- Issue #1138²⁵⁸⁵ - clean up pkg-config files
- Issue #1137²⁵⁸⁶ - Improvements to create binary packages
- Issue #1136²⁵⁸⁷ - `HPX_GCC_VERSION` not defined on all compilers
- Issue #1135²⁵⁸⁸ - Avoiding collision between `winsock2.h` and `windows.h`
- Issue #1134²⁵⁸⁹ - Making sure, that `hpx::finalize` can be called from any locality
- Issue #1133²⁵⁹⁰ - `1d stencil` examples
- Issue #1131²⁵⁹¹ - Refactor `unique_function` implementation
- Issue #1130²⁵⁹² - Unique function

²⁵⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1154>

²⁵⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1153>

²⁵⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1152>

²⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1151>

²⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1150>

²⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1149>

²⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1148>

²⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1147>

²⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1146>

²⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1145>

²⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1144>

²⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1143>

²⁵⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1142>

²⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1140>

²⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1139>

²⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1138>

²⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1137>

²⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1136>

²⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1135>

²⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1134>

²⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1133>

²⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1131>

²⁵⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1130>

- Issue #1129²⁵⁹³ - Some fixes to the Build system on OS X
- Issue #1128²⁵⁹⁴ - Action future args
- Issue #1127²⁵⁹⁵ - Executor causes segmentation fault
- Issue #1124²⁵⁹⁶ - Adding new API functions: `register_id_with_basename`, `unregister_id_with_basename`, `find_ids_from_basename`; adding test
- Issue #1123²⁵⁹⁷ - Reduce nesting of try-catch construct in `encode_parcels`?
- Issue #1122²⁵⁹⁸ - Client base fixes
- Issue #1121²⁵⁹⁹ - Update `hpxrun.py.in`
- Issue #1120²⁶⁰⁰ - HTTPS2 tests compile errors on v110 (VS2012)
- Issue #1119²⁶⁰¹ - Remove references to `boost::atomic` in accumulator example
- Issue #1118²⁶⁰² - Only build test `thread_pool_executor_1114_test` if `HPX_LOCAL_SCHEDULER` is set
- Issue #1117²⁶⁰³ - `local_queue_executor` linker error on vc110
- Issue #1116²⁶⁰⁴ - Disabled performance counter should give runtime errors, not invalid data
- Issue #1115²⁶⁰⁵ - Compile error with Intel C++ 13.1
- Issue #1114²⁶⁰⁶ - Default constructed executor is not usable
- Issue #1113²⁶⁰⁷ - Fast compilation of logging causes ABI incompatibilities between different `NDEBUG` values
- Issue #1112²⁶⁰⁸ - Using `thread_pool_executors` causes segfault
- Issue #1111²⁶⁰⁹ - `hpx::threads::get_thread_data` always returns zero
- Issue #1110²⁶¹⁰ - Remove unnecessary null pointer checks
- Issue #1109²⁶¹¹ - More tests adjustments
- Issue #1108²⁶¹² - Clarify build rules for “`libboost_atomic-mt.so`”?
- Issue #1107²⁶¹³ - Remove unnecessary null pointer checks
- Issue #1106²⁶¹⁴ - `network_storage` benchmark improvements, adding legends to plots and tidying layout
- Issue #1105²⁶¹⁵ - Add more plot outputs and improve instructions doc

²⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1129>

²⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1128>

²⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1127>

²⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1124>

²⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1123>

²⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1122>

²⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1121>

²⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1120>

²⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1119>

²⁶⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1118>

²⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1117>

²⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1116>

²⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1115>

²⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1114>

²⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1113>

²⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1112>

²⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1111>

²⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1110>

²⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1109>

²⁶¹² <https://github.com/STELLAR-GROUP/hpx/issues/1108>

²⁶¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1107>

²⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1106>

²⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1105>

- Issue #1104²⁶¹⁶ - Complete quoting for parameters of some CMake commands
- Issue #1103²⁶¹⁷ - Work on test/scripts
- Issue #1102²⁶¹⁸ - Changed minimum requirement of window install to 2012
- Issue #1101²⁶¹⁹ - Changed minimum requirement of window install to 2012
- Issue #1100²⁶²⁰ - Changed readme to no longer specify using MSVC 2010 compiler
- Issue #1099²⁶²¹ - Error returning futures from component actions
- Issue #1098²⁶²² - Improve storage test
- Issue #1097²⁶²³ - data_actions quickstart example calls missing function decorate_action of data_get_action
- Issue #1096²⁶²⁴ - MPI parcelport broken with new zero copy optimization
- Issue #1095²⁶²⁵ - Warning C4005: _WIN32_WINNT: Macro redefinition
- Issue #1094²⁶²⁶ - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
- Issue #1093²⁶²⁷ - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS
- Issue #1092²⁶²⁸ - Rename unique_future<> back to future<>
- Issue #1091²⁶²⁹ - Inconsistent error message
- Issue #1090²⁶³⁰ - On windows 8.1 the examples crashed if using more than one os thread
- Issue #1089²⁶³¹ - Components should be allowed to have their own executor
- Issue #1088²⁶³² - Add possibility to select a network interface for the ibverbs parcelport
- Issue #1087²⁶³³ - ibverbs and ipc parcelport uses zero copy optimization
- Issue #1083²⁶³⁴ - Make shell examples copyable in docs
- Issue #1082²⁶³⁵ - Implement proper termination detection during shutdown
- Issue #1081²⁶³⁶ - Implement thread_specific_ptr for hpx::threads
- Issue #1072²⁶³⁷ - make install not working properly
- Issue #1070²⁶³⁸ - Complete quoting for parameters of some CMake commands

²⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1104>

²⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1103>

²⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1102>

²⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1101>

²⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1100>

²⁶²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1099>

²⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/1098>

²⁶²³ <https://github.com/STELLAR-GROUP/hpx/issues/1097>

²⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1096>

²⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1095>

²⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1094>

²⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1093>

²⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1092>

²⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1091>

²⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1090>

²⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1089>

²⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/1088>

²⁶³³ <https://github.com/STELLAR-GROUP/hpx/issues/1087>

²⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1083>

²⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1082>

²⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1081>

²⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1072>

²⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1070>

- Issue #1059²⁶³⁹ - Fix more unused variable warnings
- Issue #1051²⁶⁴⁰ - Implement when_each
- Issue #973²⁶⁴¹ - Would like option to report hwloc bindings
- Issue #970²⁶⁴² - Bad flags for Fortran compiler
- Issue #941²⁶⁴³ - Create a proper user level context switching class for BG/Q
- Issue #935²⁶⁴⁴ - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #934²⁶⁴⁵ - Want to build HPX without dynamic libraries
- Issue #927²⁶⁴⁶ - Make hpx/lcos/reduce.hpp accept futures of id_type
- Issue #926²⁶⁴⁷ - All unit tests that are run with more than one thread with CTest/hpx_run_test should configure hpx.os_threads
- Issue #925²⁶⁴⁸ - regression_dataflow_791 needs to be brought in line with HPX standards
- Issue #899²⁶⁴⁹ - Fix race conditions in regression tests
- Issue #879²⁶⁵⁰ - Hung test leads to cascading test failure; make tests should support the MPI parcelport
- Issue #865²⁶⁵¹ - future<T> and friends shall work for movable only Ts
- Issue #847²⁶⁵² - Dynamic libraries are not installed on OS X
- Issue #816²⁶⁵³ - First Program tutorial pull request
- Issue #799²⁶⁵⁴ - Wrap lexical_cast to avoid exceptions
- Issue #720²⁶⁵⁵ - broken configuration when using cmake on Ubuntu
- Issue #622²⁶⁵⁶ - --hpx:hpx and --hpx:debug-hpx-log is nonsensical
- Issue #525²⁶⁵⁷ - Extend barrier LCO test to run in distributed
- Issue #515²⁶⁵⁸ - Multi-destination version of hpx::apply is broken
- Issue #509²⁶⁵⁹ - Push Boost.Atomic changes upstream
- Issue #503²⁶⁶⁰ - Running HPX applications on Windows should not require setting %PATH%
- Issue #461²⁶⁶¹ - Add a compilation sanity test

²⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1059>

²⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1051>

²⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/973>

²⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/970>

²⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/941>

²⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/935>

²⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/934>

²⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/927>

²⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/926>

²⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/925>

²⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/899>

²⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/879>

²⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/865>

²⁶⁵² <https://github.com/STELLAR-GROUP/hpx/issues/847>

²⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/816>

²⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/799>

²⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/720>

²⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/622>

²⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/525>

²⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/515>

²⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/509>

²⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/503>

²⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/461>

- [Issue #456](#)²⁶⁶² - `hpx_run_tests.py` should log output from tests that timeout
- [Issue #454](#)²⁶⁶³ - Investigate threadmanager performance
- [Issue #345](#)²⁶⁶⁴ - Add more versatile environmental/cmake variable support to `hpx_find_*` CMake macros
- [Issue #209](#)²⁶⁶⁵ - Support multiple configurations in generated build files
- [Issue #190](#)²⁶⁶⁶ - `hpx::cout` should be a `std::ostream`
- [Issue #189](#)²⁶⁶⁷ - `iostreams` component should use startup/shutdown functions
- [Issue #183](#)²⁶⁶⁸ - Use Boost.ICL for correctness in AGAS
- [Issue #44](#)²⁶⁶⁹ - 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.
- 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 Standard²⁶⁷⁰. While `hpx::shared_future` is new and will not create any compatibility problems, we revised the interface

²⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/456>

²⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/454>

²⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/345>

²⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/209>

²⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/190>

²⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/189>

²⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/183>

²⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/44>

²⁶⁷⁰ <http://www.open-std.org/jtc1/sc22/wg21>

and implementation of the existing `hpx::future`. For more details please see the [mailing list archive](#)²⁶⁷¹. 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 Standard²⁶⁷².

- 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](#)²⁶⁷³.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- [Issue #1086](#)²⁶⁷⁴ - Expose internal `boost::shared_array` to allow user management of array lifetime
- [Issue #1083](#)²⁶⁷⁵ - Make shell examples copyable in docs
- [Issue #1080](#)²⁶⁷⁶ - `/threads{locality#*/total}/count/cumulative` broken
- [Issue #1079](#)²⁶⁷⁷ - Build problems on OS X
- [Issue #1078](#)²⁶⁷⁸ - Improve robustness of component loading
- [Issue #1077](#)²⁶⁷⁹ - Fix a missing enum definition for 'take' mode

²⁶⁷¹ <http://mail.cct.lsu.edu/pipermail/hpx-users/2014-January/000141.html>

²⁶⁷² <http://www.open-std.org/jtc1/sc22/wg21>

²⁶⁷³ <https://github.com/STELLAR-GROUP/hpxpi/blob/master/spec.pdf?raw=true>

²⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1086>

²⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1083>

²⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1080>

²⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1079>

²⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1078>

²⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1077>

- Issue #1076²⁶⁸⁰ - Merge Jb master
- Issue #1075²⁶⁸¹ - Unknown CMake command “add_hpx_pseudo_target”
- Issue #1074²⁶⁸² - Implement `apply_continue_callback` and `apply_colocated_callback`
- Issue #1073²⁶⁸³ - The new `apply_colocated` and `async_colocated` functions lead to automatic registered functions
- Issue #1071²⁶⁸⁴ - Remove `deferred_packaged_task`
- Issue #1069²⁶⁸⁵ - `serialize_buffer` with allocator fails at destruction
- Issue #1068²⁶⁸⁶ - Coroutine include and forward declarations missing
- Issue #1067²⁶⁸⁷ - Add allocator support to `util::serialize_buffer`
- Issue #1066²⁶⁸⁸ - Allow for `MPI_Init` being called before HPX launches
- Issue #1065²⁶⁸⁹ - AGAS cache isn't used/populated on worker localities
- Issue #1064²⁶⁹⁰ - Reorder includes to ensure `ws2` includes early
- Issue #1063²⁶⁹¹ - Add `hpx::runtime::suspend` and `hpx::runtime::resume`
- Issue #1062²⁶⁹² - Fix `async_continue` to properly handle return types
- Issue #1061²⁶⁹³ - Implement `async_colocated` and `apply_colocated`
- Issue #1060²⁶⁹⁴ - Implement minimal component migration
- Issue #1058²⁶⁹⁵ - Remove `HPX_UTIL_TUPLE` from code base
- Issue #1057²⁶⁹⁶ - Add performance counters for threading subsystem
- Issue #1055²⁶⁹⁷ - Thread allocation uses two memory pools
- Issue #1053²⁶⁹⁸ - Work stealing flawed
- Issue #1052²⁶⁹⁹ - Fix a number of warnings
- Issue #1049²⁷⁰⁰ - Fixes for TLS on OSX and more reliable test running
- Issue #1048²⁷⁰¹ - Fixing after 588 hang
- Issue #1047²⁷⁰² - Use port '0' for networking when using one locality

²⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1076>

²⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1075>

²⁶⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1074>

²⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1073>

²⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1071>

²⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1069>

²⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1068>

²⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1067>

²⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1066>

²⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1065>

²⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1064>

²⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1063>

²⁶⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1062>

²⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1061>

²⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1060>

²⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1058>

²⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1057>

²⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1055>

²⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1053>

²⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1052>

²⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1049>

²⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1048>

²⁷⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1047>

- Issue #1046²⁷⁰³ - `composable_guard` test is broken when having more than one thread
- Issue #1045²⁷⁰⁴ - Security missing headers
- Issue #1044²⁷⁰⁵ - Native TLS on FreeBSD via `__thread`
- Issue #1043²⁷⁰⁶ - `async` et.al. compute the wrong result type
- Issue #1042²⁷⁰⁷ - `async` et.al. implicitly unwrap `reference_wrappers`
- Issue #1041²⁷⁰⁸ - Remove redundant costly Kleene stars from regex searches
- Issue #1040²⁷⁰⁹ - CMake script regex match patterns has unnecessary kleenes
- Issue #1039²⁷¹⁰ - Remove use of `Boost.Move` and replace with `std::move` and real rvalue refs
- Issue #1038²⁷¹¹ - Bump minimal required Boost to 1.49.0
- Issue #1037²⁷¹² - Implicit unwrapping of futures in `async` broken
- Issue #1036²⁷¹³ - Scheduler hangs when user code attempts to “block” OS-threads
- Issue #1035²⁷¹⁴ - Idle-rate counter always reports 100% idle rate
- Issue #1034²⁷¹⁵ - Symbolic name registration causes application hangs
- Issue #1033²⁷¹⁶ - Application options read in from an options file generate an error message
- Issue #1032²⁷¹⁷ - `hpx::id_type` local reference counting is wrong
- Issue #1031²⁷¹⁸ - Negative entry in reference count table
- Issue #1030²⁷¹⁹ - Implement `condition_variable`
- Issue #1029²⁷²⁰ - Deadlock in thread scheduling subsystem
- Issue #1028²⁷²¹ - HPX-thread cumulative count performance counters report incorrect value
- Issue #1027²⁷²² - Expose `hpx::thread_interrupted` error code as a separate exception type
- Issue #1026²⁷²³ - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
- Issue #1025²⁷²⁴ - `future::wait_for/wait_until` do not remove callback
- Issue #1024²⁷²⁵ - Remove dependence to boost assert and create `hpx` assert

²⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1046>

²⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1045>

²⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1044>

²⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1043>

²⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1042>

²⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1041>

²⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1040>

²⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1039>

²⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1038>

²⁷¹² <https://github.com/STELLAR-GROUP/hpx/issues/1037>

²⁷¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1036>

²⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1035>

²⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1034>

²⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1033>

²⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1032>

²⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1031>

²⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1030>

²⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1029>

²⁷²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1028>

²⁷²² <https://github.com/STELLAR-GROUP/hpx/issues/1027>

²⁷²³ <https://github.com/STELLAR-GROUP/hpx/issues/1026>

²⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1025>

²⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1024>

- Issue #1023²⁷²⁶ - Segfaults with tcmalloc
- Issue #1022²⁷²⁷ - prerequisites link in readme is broken
- Issue #1020²⁷²⁸ - HPX Deadlock on external synchronization
- Issue #1019²⁷²⁹ - Convert using BOOST_ASSERT to HPX_ASSERT
- Issue #1018²⁷³⁰ - compiling bug with gcc 4.8.1
- Issue #1017²⁷³¹ - Possible crash in io_pool executor
- Issue #1016²⁷³² - Crash at startup
- Issue #1014²⁷³³ - Implement Increment/Decrement Merging
- Issue #1013²⁷³⁴ - Add more logging channels to enable greater control over logging granularity
- Issue #1012²⁷³⁵ - --hpx:debug-hpx-log and --hpx:debug-agas-log lead to non-thread safe writes
- Issue #1011²⁷³⁶ - After installation, running applications from the build/staging directory no longer works
- Issue #1010²⁷³⁷ - Mergable decrement requests are not being merged
- Issue #1009²⁷³⁸ - --hpx:list-symbolic-names crashes
- Issue #1007²⁷³⁹ - Components are not properly destroyed
- Issue #1006²⁷⁴⁰ - Segfault/hang in set_data
- Issue #1003²⁷⁴¹ - Performance counter naming issue
- Issue #982²⁷⁴² - Race condition during startup
- Issue #912²⁷⁴³ - OS X: component type not found in map
- Issue #663²⁷⁴⁴ - Create a buildbot slave based on Clang 3.2/OSX
- Issue #636²⁷⁴⁵ - Expose this_locality::apply<act>(p1, p2); for local execution
- Issue #197²⁷⁴⁶ - Add --console=address option for PBS runs
- Issue #175²⁷⁴⁷ - Asynchronous AGAS API

²⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1023>

²⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1022>

²⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1020>

²⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1019>

²⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1018>

²⁷³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1017>

²⁷³² <https://github.com/STELLAR-GROUP/hpx/issues/1016>

²⁷³³ <https://github.com/STELLAR-GROUP/hpx/issues/1014>

²⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1013>

²⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1012>

²⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1011>

²⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1010>

²⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1009>

²⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1007>

²⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1006>

²⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1003>

²⁷⁴² <https://github.com/STELLAR-GROUP/hpx/issues/982>

²⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/912>

²⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/663>

²⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/636>

²⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/197>

²⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/175>

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 #1005²⁷⁴⁸](#) - Allow to disable array optimizations and zero copy optimizations for each parcelport
- [Issue #1004²⁷⁴⁹](#) - Generate new HPX logo image for the docs
- [Issue #1002²⁷⁵⁰](#) - If MPI parcelport is not available, running HPX under mpirun should fail
- [Issue #1001²⁷⁵¹](#) - Zero copy serialization raises assert
- [Issue #1000²⁷⁵²](#) - Can't connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
- [Issue #999²⁷⁵³](#) - Optimize `hpx::when_n`

²⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1005>

²⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1004>

²⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1002>

²⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1001>

²⁷⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1000>

²⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/999>

- [Issue #998²⁷⁵⁴](#) - Fixed const-correctness
- [Issue #997²⁷⁵⁵](#) - Making `serialize_buffer::data()` type save
- [Issue #996²⁷⁵⁶](#) - Memory leak in `hpx::lcos::promise`
- [Issue #995²⁷⁵⁷](#) - Race while registering pre-shutdown functions
- [Issue #994²⁷⁵⁸](#) - `thread_rescheduling` regression test does not compile
- [Issue #992²⁷⁵⁹](#) - Correct comments and messages
- [Issue #991²⁷⁶⁰](#) - `setcap cap_sys_rawio=ep` for power profiling causes an HPX application to abort
- [Issue #989²⁷⁶¹](#) - Jacobi hangs during execution
- [Issue #988²⁷⁶²](#) - `multiple_init` test is failing
- [Issue #986²⁷⁶³](#) - Can't call a function called "init" from "main" when using `<hpx/hpx_main.hpp>`
- [Issue #984²⁷⁶⁴](#) - Reference counting tests are failing
- [Issue #983²⁷⁶⁵](#) - `thread_suspension_executor` test fails
- [Issue #980²⁷⁶⁶](#) - Terminating HPX threads don't leave stack in virgin state
- [Issue #979²⁷⁶⁷](#) - Static scheduler not in documents
- [Issue #978²⁷⁶⁸](#) - Preprocessing limits are broken
- [Issue #977²⁷⁶⁹](#) - Make `tests.regressions.lcos.future_hang_on_get` shorter
- [Issue #976²⁷⁷⁰](#) - Wrong library order in `pkgconfig`
- [Issue #975²⁷⁷¹](#) - Please reopen #963
- [Issue #974²⁷⁷²](#) - Option `pu-offset` ignored in `fixing_588` branch
- [Issue #972²⁷⁷³](#) - Cannot use MKL with HPX
- [Issue #969²⁷⁷⁴](#) - Non-existent INI files requested on the command line via `--hpx:config` do not cause warnings or errors.
- [Issue #968²⁷⁷⁵](#) - Cannot build examples in `fixing_588` branch
- [Issue #967²⁷⁷⁶](#) - Command line description of `--hpx:queuing` seems wrong

²⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/998>

²⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/997>

²⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/996>

²⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/995>

²⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/994>

²⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/992>

²⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/991>

²⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/989>

²⁷⁶² <https://github.com/STELLAR-GROUP/hpx/issues/988>

²⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/986>

²⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/984>

²⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/983>

²⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/980>

²⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/979>

²⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/978>

²⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/977>

²⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/976>

²⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/975>

²⁷⁷² <https://github.com/STELLAR-GROUP/hpx/issues/974>

²⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/972>

²⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/969>

²⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/968>

²⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/967>

- Issue #966²⁷⁷⁷ - `--hpx:print-bind` physical core numbers are wrong
- Issue #965²⁷⁷⁸ - Deadlock when building in Release mode
- Issue #963²⁷⁷⁹ - Not all worker threads are working
- Issue #962²⁷⁸⁰ - Problem with SLURM integration
- Issue #961²⁷⁸¹ - `--hpx:print-bind` outputs incorrect information
- Issue #960²⁷⁸² - Fix cut and paste error in documentation of `get_thread_priority`
- Issue #959²⁷⁸³ - Change link to `boost.atomic` in documentation to point to `boost.org`
- Issue #958²⁷⁸⁴ - Undefined reference to `intrusive_ptr_release`
- Issue #957²⁷⁸⁵ - Make tuple standard compliant
- Issue #956²⁷⁸⁶ - Segfault with `a3382fb`
- Issue #955²⁷⁸⁷ - `--hpx:nodes` and `--hpx:nodefiles` do not work with foreign nodes
- Issue #954²⁷⁸⁸ - Make order of arguments for `hpx::async` and `hpx::broadcast` consistent
- Issue #953²⁷⁸⁹ - Cannot use MKL with HPX
- Issue #952²⁷⁹⁰ - `register_[pre_]shutdown_function` never throw
- Issue #951²⁷⁹¹ - Assert when number of threads is greater than hardware concurrency
- Issue #948²⁷⁹² - `HPX_HAVE_GENERIC_CONTEXT_COROUTINES` conflicts with `HPX_HAVE_FIBER_BASED_COROUTINES`
- Issue #947²⁷⁹³ - Need `MPI_THREAD_MULTIPLE` for backward compatibility
- Issue #946²⁷⁹⁴ - HPX does not call `MPI_Finalize`
- Issue #945²⁷⁹⁵ - Segfault with `hpx::lcos::broadcast`
- Issue #944²⁷⁹⁶ - OS X: assertion `pu_offset_ < hardware_concurrency` failed
- Issue #943²⁷⁹⁷ - `#include <hpx/hpx_main.hpp>` does not work
- Issue #942²⁷⁹⁸ - Make the BG/Q work with `-O3`
- Issue #940²⁷⁹⁹ - Use separator when concatenating locality name

²⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/966>

²⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/965>

²⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/963>

²⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/962>

²⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/961>

²⁷⁸² <https://github.com/STELLAR-GROUP/hpx/issues/960>

²⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/959>

²⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/958>

²⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/957>

²⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/956>

²⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/955>

²⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/954>

²⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/953>

²⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/952>

²⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/951>

²⁷⁹² <https://github.com/STELLAR-GROUP/hpx/issues/948>

²⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/947>

²⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/946>

²⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/945>

²⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/944>

²⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/943>

²⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/942>

²⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/940>

- [Issue #939](#)²⁸⁰⁰ - Refactor MPI parcelport to use `MPI_Wait` instead of multiple `MPI_Test` calls
- [Issue #938](#)²⁸⁰¹ - Want to officially access `client_base::gid_`
- [Issue #937](#)²⁸⁰² - `client_base::gid_` should be private²⁸⁰³
- [Issue #936](#)²⁸⁰³ - Want doxygen-like source code index
- [Issue #935](#)²⁸⁰⁴ - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- [Issue #933](#)²⁸⁰⁵ - Cannot build HPX with Boost 1.54.0
- [Issue #932](#)²⁸⁰⁶ - Components are destructed too early
- [Issue #931](#)²⁸⁰⁷ - Make HPX work on BG/Q
- [Issue #930](#)²⁸⁰⁸ - make git-docs is broken
- [Issue #929](#)²⁸⁰⁹ - Generating index in docs broken
- [Issue #928](#)²⁸¹⁰ - Optimize `hpx::util::static_` for C++11 compilers supporting magic statics
- [Issue #924](#)²⁸¹¹ - Make `kill_process_tree` (in `process.py`) more robust on Mac OSX
- [Issue #923](#)²⁸¹² - Correct BLAS and RNPL cmake tests
- [Issue #922](#)²⁸¹³ - Cannot link against BLAS
- [Issue #921](#)²⁸¹⁴ - Implement `hpx::mem_fn`
- [Issue #920](#)²⁸¹⁵ - Output locality with `--hpx:print-bind`
- [Issue #919](#)²⁸¹⁶ - Correct grammar; simplify boolean expressions
- [Issue #918](#)²⁸¹⁷ - Link to `hello_world.cpp` is broken
- [Issue #917](#)²⁸¹⁸ - adapt cmake file to new boostbook version
- [Issue #916](#)²⁸¹⁹ - fix problem building documentation with `xsltproc` \geq 1.1.27
- [Issue #915](#)²⁸²⁰ - Add another TBBMalloc library search path
- [Issue #914](#)²⁸²¹ - Build problem with Intel compiler on Stampede (TACC)
- [Issue #913](#)²⁸²² - fix error messages in fibonacci examples

²⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/939>

²⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/938>

²⁸⁰² <https://github.com/STELLAR-GROUP/hpx/issues/937>

²⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/936>

²⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/935>

²⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/933>

²⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/932>

²⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/931>

²⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/930>

²⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/929>

²⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/928>

²⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/924>

²⁸¹² <https://github.com/STELLAR-GROUP/hpx/issues/923>

²⁸¹³ <https://github.com/STELLAR-GROUP/hpx/issues/922>

²⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/921>

²⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/920>

²⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/919>

²⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/918>

²⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/917>

²⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/916>

²⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/915>

²⁸²¹ <https://github.com/STELLAR-GROUP/hpx/issues/914>

²⁸²² <https://github.com/STELLAR-GROUP/hpx/issues/913>

- Issue #911²⁸²³ - Update OS X build instructions
- Issue #910²⁸²⁴ - Want like to specify MPI_ROOT instead of compiler wrapper script
- Issue #909²⁸²⁵ - Warning about void* arithmetic
- Issue #908²⁸²⁶ - Buildbot for MIC is broken
- Issue #906²⁸²⁷ - Can't use --hpx:bind=balanced with multiple MPI processes
- Issue #905²⁸²⁸ - --hpx:bind documentation should describe full grammar
- Issue #904²⁸²⁹ - Add hpx::lcos::fold and hpx::lcos::inverse_fold collective operation
- Issue #903²⁸³⁰ - Add hpx::when_any_swapped()
- Issue #902²⁸³¹ - Add hpx::lcos::reduce collective operation
- Issue #901²⁸³² - Web documentation is not searchable
- Issue #900²⁸³³ - Web documentation for trunk has no index
- Issue #898²⁸³⁴ - Some tests fail with GCC 4.8.1 and MPI parcel port
- Issue #897²⁸³⁵ - HWLOC causes failures on Mac
- Issue #896²⁸³⁶ - pu-offset leads to startup error
- Issue #895²⁸³⁷ - hpx::get_locality_name not defined
- Issue #894²⁸³⁸ - Race condition at shutdown
- Issue #893²⁸³⁹ - --hpx:print-bind switches std::cout to hexadecimal mode
- Issue #892²⁸⁴⁰ - hwloc_topology_load can be expensive – don't call multiple times
- Issue #891²⁸⁴¹ - The documentation for get_locality_name is wrong
- Issue #890²⁸⁴² - --hpx:print-bind should not exit
- Issue #889²⁸⁴³ - --hpx:debug-hpx-log=FILE does not work
- Issue #888²⁸⁴⁴ - MPI parcelport does not exit cleanly for --hpx:print-bind
- Issue #887²⁸⁴⁵ - Choose thread affinities more cleverly

²⁸²³ <https://github.com/STELLAR-GROUP/hpx/issues/911>

²⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/910>

²⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/909>

²⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/908>

²⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/906>

²⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/905>

²⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/904>

²⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/903>

²⁸³¹ <https://github.com/STELLAR-GROUP/hpx/issues/902>

²⁸³² <https://github.com/STELLAR-GROUP/hpx/issues/901>

²⁸³³ <https://github.com/STELLAR-GROUP/hpx/issues/900>

²⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/898>

²⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/897>

²⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/896>

²⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/895>

²⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/894>

²⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/893>

²⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/892>

²⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/891>

²⁸⁴² <https://github.com/STELLAR-GROUP/hpx/issues/890>

²⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/889>

²⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/888>

²⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/887>

- [Issue #886](#)²⁸⁴⁶ - Logging documentation is confusing
- [Issue #885](#)²⁸⁴⁷ - Two threads are slower than one
- [Issue #884](#)²⁸⁴⁸ - `is_callable` failing with member pointers in C++11
- [Issue #883](#)²⁸⁴⁹ - Need help with `is_callable_test`
- [Issue #882](#)²⁸⁵⁰ - `tests.regressions.lcos.future_hang_on_get` does not terminate
- [Issue #881](#)²⁸⁵¹ - `tests.regressions/block_matrix/matrix.hh` won't compile with GCC 4.8.1
- [Issue #880](#)²⁸⁵² - HPX does not work on OS X
- [Issue #878](#)²⁸⁵³ - `future::unwrap` triggers assertion
- [Issue #877](#)²⁸⁵⁴ - “make tests” has build errors on Ubuntu 12.10
- [Issue #876](#)²⁸⁵⁵ - `tcmalloc` is used by default, even if it is not present
- [Issue #875](#)²⁸⁵⁶ - `global_fixture` is defined in a header file
- [Issue #874](#)²⁸⁵⁷ - Some tests take very long
- [Issue #873](#)²⁸⁵⁸ - Add block-matrix code as regression test
- [Issue #872](#)²⁸⁵⁹ - HPX documentation does not say how to run tests with detailed output
- [Issue #871](#)²⁸⁶⁰ - All tests fail with “make test”
- [Issue #870](#)²⁸⁶¹ - Please explicitly disable serialization in classes that don't support it
- [Issue #868](#)²⁸⁶² - `boost_any` test failing
- [Issue #867](#)²⁸⁶³ - Reduce the number of copies of `hpx::function` arguments
- [Issue #863](#)²⁸⁶⁴ - Futures should not require a default constructor
- [Issue #862](#)²⁸⁶⁵ - `value_or_error` shall not default construct its result
- [Issue #861](#)²⁸⁶⁶ - `HPX_UNUSED` macro
- [Issue #860](#)²⁸⁶⁷ - Add functionality to copy construct a component
- [Issue #859](#)²⁸⁶⁸ - `hpx::endl` should flush

²⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/886>

²⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/885>

²⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/884>

²⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/883>

²⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/882>

²⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/881>

²⁸⁵² <https://github.com/STELLAR-GROUP/hpx/issues/880>

²⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/878>

²⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/877>

²⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/876>

²⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/875>

²⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/874>

²⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/873>

²⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/872>

²⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/871>

²⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/870>

²⁸⁶² <https://github.com/STELLAR-GROUP/hpx/issues/868>

²⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/867>

²⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/863>

²⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/862>

²⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/861>

²⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/860>

²⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/859>

- Issue #858²⁸⁶⁹ - Create `hpx::get_ptr<>` allowing to access component implementation
- Issue #855²⁸⁷⁰ - Implement `hpx::INVOKE`
- Issue #854²⁸⁷¹ - `hpx/hpx.hpp` does not include `hpx/include/iostreams.hpp`
- Issue #853²⁸⁷² - Feature request: null future
- Issue #852²⁸⁷³ - Feature request: Locality names
- Issue #851²⁸⁷⁴ - `hpx::cout` output does not appear on screen
- Issue #849²⁸⁷⁵ - All tests fail on OS X after installing
- Issue #848²⁸⁷⁶ - Update OS X build instructions
- Issue #846²⁸⁷⁷ - Update `hpx_external_example`
- Issue #845²⁸⁷⁸ - Issues with having both debug and release modules in the same directory
- Issue #844²⁸⁷⁹ - Create configuration header
- Issue #843²⁸⁸⁰ - Tests should use CTest
- Issue #842²⁸⁸¹ - Remove `buffer_pool` from MPI parcellport
- Issue #841²⁸⁸² - Add possibility to broadcast an index with `hpx::lcos::broadcast`
- Issue #838²⁸⁸³ - Simplify `util::tuple`
- Issue #837²⁸⁸⁴ - Adopt `boost::tuple` tests for `util::tuple`
- Issue #836²⁸⁸⁵ - Adopt `boost::function` tests for `util::function`
- Issue #835²⁸⁸⁶ - Tuple interface missing pieces
- Issue #833²⁸⁸⁷ - Partially preprocessing files not working
- Issue #832²⁸⁸⁸ - Native papi counters do not work with wild cards
- Issue #831²⁸⁸⁹ - Arithmetics counter fails if only one parameter is given
- Issue #830²⁸⁹⁰ - Convert `hpx::util::function` to use new scheme for serializing its base pointer
- Issue #829²⁸⁹¹ - Consistently use `decay<T>` instead of `remove_const< remove_reference<T>>`

²⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/858>

²⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/855>

²⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/854>

²⁸⁷² <https://github.com/STELLAR-GROUP/hpx/issues/853>

²⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/852>

²⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/851>

²⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/849>

²⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/848>

²⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/846>

²⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/845>

²⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/844>

²⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/843>

²⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/842>

²⁸⁸² <https://github.com/STELLAR-GROUP/hpx/issues/841>

²⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/838>

²⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/837>

²⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/836>

²⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/835>

²⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/833>

²⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/832>

²⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/831>

²⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/830>

²⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/829>

- [Issue #828](#)²⁸⁹² - Update future implementation to N3721 and N3722
- [Issue #827](#)²⁸⁹³ - Enable MPI parcelport for bootstrapping whenever application was started using mpirun
- [Issue #826](#)²⁸⁹⁴ - Support command line option `--hpx:print-bind` even if `--hpx::bind` was not used
- [Issue #825](#)²⁸⁹⁵ - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- [Issue #824](#)²⁸⁹⁶ - Enable lambda functions to be used with `hpx::async/hpx::apply`
- [Issue #823](#)²⁸⁹⁷ - Using a hashing filter
- [Issue #822](#)²⁸⁹⁸ - Silence unused variable warning
- [Issue #821](#)²⁸⁹⁹ - Detect if a function object is callable with given arguments
- [Issue #820](#)²⁹⁰⁰ - Allow wildcards to be used for performance counter names
- [Issue #819](#)²⁹⁰¹ - Make the AGAS symbolic name registry distributed
- [Issue #818](#)²⁹⁰² - Add `future::then()` overload taking an executor
- [Issue #817](#)²⁹⁰³ - Fixed typo
- [Issue #815](#)²⁹⁰⁴ - Create an lco that is performing an efficient broadcast of actions
- [Issue #814](#)²⁹⁰⁵ - Papi counters cannot specify `thread#*` to get the counts for all threads
- [Issue #813](#)²⁹⁰⁶ - Scoped unlock
- [Issue #811](#)²⁹⁰⁷ - `simple_central_tuplespace_client` run error
- [Issue #810](#)²⁹⁰⁸ - ostream error when `<<` any objects
- [Issue #809](#)²⁹⁰⁹ - Optimize parcel serialization
- [Issue #808](#)²⁹¹⁰ - HPX applications throw exception when executed from the build directory
- [Issue #807](#)²⁹¹¹ - Create performance counters exposing overall AGAS statistics
- [Issue #795](#)²⁹¹² - Create timed `make_ready_future`
- [Issue #794](#)²⁹¹³ - Create heterogeneous `when_all/when_any/etc.`
- [Issue #721](#)²⁹¹⁴ - Make HPX usable for Xeon Phi

²⁸⁹² <https://github.com/STELLAR-GROUP/hpx/issues/828>

²⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/827>

²⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/826>

²⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/825>

²⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/824>

²⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/823>

²⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/822>

²⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/821>

²⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/820>

²⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/819>

²⁹⁰² <https://github.com/STELLAR-GROUP/hpx/issues/818>

²⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/817>

²⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/815>

²⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/814>

²⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/813>

²⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/811>

²⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/810>

²⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/809>

²⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/808>

²⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/807>

²⁹¹² <https://github.com/STELLAR-GROUP/hpx/issues/795>

²⁹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/794>

²⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/721>

- [Issue #694](#)²⁹¹⁵ - CMake should complain if you attempt to build an example without its dependencies
- [Issue #692](#)²⁹¹⁶ - SLURM support broken
- [Issue #683](#)²⁹¹⁷ - python/hpx/process.py imports epoll on all platforms
- [Issue #619](#)²⁹¹⁸ - Automate the doc building process
- [Issue #600](#)²⁹¹⁹ - GTC performance broken
- [Issue #577](#)²⁹²⁰ - Allow for zero copy serialization/networking
- [Issue #551](#)²⁹²¹ - Change executable names to have debug postfix in Debug builds
- [Issue #544](#)²⁹²² - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init
- [Issue #534](#)²⁹²³ - hpx::init should take functions by std::function and should accept all forms of hpx_main
- [Issue #508](#)²⁹²⁴ - FindPackage fails to set FOO_LIBRARY_DIR
- [Issue #506](#)²⁹²⁵ - Add cmake support to generate ini files for external applications
- [Issue #470](#)²⁹²⁶ - Changing build-type after configure does not update boost library names
- [Issue #453](#)²⁹²⁷ - Document hpx_run_tests.py
- [Issue #445](#)²⁹²⁸ - Significant performance mismatch between MPI and HPX in SMP for allgather example
- [Issue #443](#)²⁹²⁹ - Make docs viewable from build directory
- [Issue #421](#)²⁹³⁰ - Support multiple HPX instances per node in a batch environment like PBS or SLURM
- [Issue #316](#)²⁹³¹ - Add message size limitation
- [Issue #249](#)²⁹³² - Clean up locking code in big boot barrier
- [Issue #136](#)²⁹³³ - 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.).

²⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/694>

²⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/692>

²⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/683>

²⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/619>

²⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/600>

²⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/577>

²⁹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/551>

²⁹²² <https://github.com/STELLAR-GROUP/hpx/issues/544>

²⁹²³ <https://github.com/STELLAR-GROUP/hpx/issues/534>

²⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/508>

²⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/506>

²⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/470>

²⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/453>

²⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/445>

²⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/443>

²⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/421>

²⁹³¹ <https://github.com/STELLAR-GROUP/hpx/issues/316>

²⁹³² <https://github.com/STELLAR-GROUP/hpx/issues/249>

²⁹³³ <https://github.com/STELLAR-GROUP/hpx/issues/136>

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 Standard²⁹³⁴ and related proposals to the C++ standardization committee (such as N3632²⁹³⁵ and N3857²⁹³⁶).
- We implemented a first version of a distributed AGAS service which essentially eliminates all explicit AGAS network traffic.
- We created a native ibverbs parcellport 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 N3562²⁹³⁷. This API is bound to change in the future, though.
- Added minimal security support for localities which is enforced on the parcellport level. This support is preliminary and experimental and might change in the future.
- We created a parcellport 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*²⁹³⁸).

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 #806²⁹³⁹ - make (all) in examples folder does nothing
- Issue #805²⁹⁴⁰ - Adding the introduction and fixing DOCBOK dependencies for Windows use
- Issue #804²⁹⁴¹ - Add stackless (non-suspendable) thread type
- Issue #803²⁹⁴² - Create proper serialization support functions for util::tuple

²⁹³⁴ <http://www.open-std.org/jtc1/sc22/wg21>

²⁹³⁵ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3632.html>

²⁹³⁶ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf>

²⁹³⁷ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf>

²⁹³⁸ [https://en.wikipedia.org/wiki/Linda_\(coordination_language\)](https://en.wikipedia.org/wiki/Linda_(coordination_language))

²⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/806>

²⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/805>

²⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/804>

²⁹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/803>

- Issue #800²⁹⁴³ - Add possibility to disable array optimizations during serialization
- Issue #798²⁹⁴⁴ - HPX_LIMIT does not work for local dataflow
- Issue #797²⁹⁴⁵ - Create a parcellport which uses MPI
- Issue #796²⁹⁴⁶ - Problem with Large Numbers of Threads
- Issue #793²⁹⁴⁷ - Changing dataflow test case to hang consistently
- Issue #792²⁹⁴⁸ - CMake Error
- Issue #791²⁹⁴⁹ - Problems with local::dataflow
- Issue #790²⁹⁵⁰ - wait_for() doesn't compile
- Issue #789²⁹⁵¹ - HPX with Intel compiler segfaults
- Issue #788²⁹⁵² - Intel compiler support
- Issue #787²⁹⁵³ - Fixed SFINAEd specializations
- Issue #786²⁹⁵⁴ - Memory issues during benchmarking.
- Issue #785²⁹⁵⁵ - Create an API allowing to register external threads with HPX
- Issue #784²⁹⁵⁶ - util::plugin is throwing an error when a symbol is not found
- Issue #783²⁹⁵⁷ - How does hpx::bind work?
- Issue #782²⁹⁵⁸ - Added quotes around STRING REPLACE potentially empty arguments
- Issue #781²⁹⁵⁹ - Make sure no exceptions propagate into the thread manager
- Issue #780²⁹⁶⁰ - Allow arithmetics performance counters to expand its parameters
- Issue #779²⁹⁶¹ - Test case for 778
- Issue #778²⁹⁶² - Swapping futures segfaults
- Issue #777²⁹⁶³ - hpx::lcos::details::when_xxx don't restore completion handlers
- Issue #776²⁹⁶⁴ - Compiler chokes on dataflow overload with launch policy
- Issue #775²⁹⁶⁵ - Runtime error with local dataflow (copying futures?)

²⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/800>

²⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/798>

²⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/797>

²⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/796>

²⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/793>

²⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/792>

²⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/791>

²⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/790>

²⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/789>

²⁹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/788>

²⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/787>

²⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/786>

²⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/785>

²⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/784>

²⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/783>

²⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/782>

²⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/781>

²⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/780>

²⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/779>

²⁹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/778>

²⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/777>

²⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/776>

²⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/775>

- [Issue #774²⁹⁶⁶](#) - Using local dataflow without explicit namespace
- [Issue #773²⁹⁶⁷](#) - Local dataflow with unwrap: functor operators need to be const
- [Issue #772²⁹⁶⁸](#) - Allow (remote) actions to return a future
- [Issue #771²⁹⁶⁹](#) - Setting HPX_LIMIT gives huge boost MPL errors
- [Issue #770²⁹⁷⁰](#) - Add launch policy to (local) dataflow
- [Issue #769²⁹⁷¹](#) - Make compile time configuration information available
- [Issue #768²⁹⁷²](#) - Const correctness problem in local dataflow
- [Issue #767²⁹⁷³](#) - Add launch policies to async
- [Issue #766²⁹⁷⁴](#) - Mark data structures for optimized (array based) serialization
- [Issue #765²⁹⁷⁵](#) - Align hpx::any with N3508: Any Library Proposal (Revision 2)
- [Issue #764²⁹⁷⁶](#) - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations
- [Issue #762²⁹⁷⁷](#) - added a human readable output for the ping pong example
- [Issue #761²⁹⁷⁸](#) - Ambiguous typename when constructing derived component
- [Issue #760²⁹⁷⁹](#) - Simple components can not be derived
- [Issue #759²⁹⁸⁰](#) - make install doesn't give a complete install
- [Issue #758²⁹⁸¹](#) - Stack overflow when using locking_hook<>
- [Issue #757²⁹⁸²](#) - copy paste error; unsupported function overloading
- [Issue #756²⁹⁸³](#) - GTCX runtime issue in Gordon
- [Issue #755²⁹⁸⁴](#) - Papi counters don't work with reset and evaluate API's
- [Issue #753²⁹⁸⁵](#) - cmake bugfix and improved component action docs
- [Issue #752²⁹⁸⁶](#) - hpx simple component docs
- [Issue #750²⁹⁸⁷](#) - Add hpx::util::any
- [Issue #749²⁹⁸⁸](#) - Thread phase counter is not reset

²⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/774>

²⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/773>

²⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/772>

²⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/771>

²⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/770>

²⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/769>

²⁹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/768>

²⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/767>

²⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/766>

²⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/765>

²⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/764>

²⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/762>

²⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/761>

²⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/760>

²⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/759>

²⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/758>

²⁹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/757>

²⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/756>

²⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/755>

²⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/753>

²⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/752>

²⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/750>

²⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/749>

- Issue #748²⁹⁸⁹ - Memory performance counter are not registered
- Issue #747²⁹⁹⁰ - Create performance counters exposing arithmetic operations
- Issue #745²⁹⁹¹ - apply_callback needs to invoke callback when applied locally
- Issue #744²⁹⁹² - CMake fixes
- Issue #743²⁹⁹³ - Problem Building github version of HPX
- Issue #742²⁹⁹⁴ - Remove HPX_STD_BIND
- Issue #741²⁹⁹⁵ - assertion 'px != 0' failed: HPX(assertion_failure) for low numbers of OS threads
- Issue #739²⁹⁹⁶ - Performance counters do not count to the end of the program or evaluation
- Issue #738²⁹⁹⁷ - Dedicated AGAS server runs don't work; console ignores -a option.
- Issue #737²⁹⁹⁸ - Missing bind overloads
- Issue #736²⁹⁹⁹ - Performance counter wildcards do not always work
- Issue #735³⁰⁰⁰ - Create native ibverbs parcelport based on rdma operations
- Issue #734³⁰⁰¹ - Threads stolen performance counter total is incorrect
- Issue #733³⁰⁰² - Test benchmarks need to be checked and fixed
- Issue #732³⁰⁰³ - Build fails with Mac, using mac ports clang-3.3 on latest git branch
- Issue #731³⁰⁰⁴ - Add global start/stop API for performance counters
- Issue #730³⁰⁰⁵ - Performance counter values are apparently incorrect
- Issue #729³⁰⁰⁶ - Unhandled switch
- Issue #728³⁰⁰⁷ - Serialization of hpx::util::function between two localities causes seg faults
- Issue #727³⁰⁰⁸ - Memory counters on Mac OS X
- Issue #725³⁰⁰⁹ - Restore original thread priority on resume
- Issue #724³⁰¹⁰ - Performance benchmarks do not depend on main HPX libraries
- Issue #723³⁰¹¹ - [teletype]-hpx:nodes="cat \$PBS_NODEFILE" works; -hpx:nodefile=\$PBS_NODEFILE does not.[c++]

²⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/748>

²⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/747>

²⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/745>

²⁹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/744>

²⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/743>

²⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/742>

²⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/741>

²⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/739>

²⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/738>

²⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/737>

²⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/736>

³⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/735>

³⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/734>

³⁰⁰² <https://github.com/STELLAR-GROUP/hpx/issues/733>

³⁰⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/732>

³⁰⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/731>

³⁰⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/730>

³⁰⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/729>

³⁰⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/728>

³⁰⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/727>

³⁰⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/725>

³⁰¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/724>

³⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/723>

- [Issue #722](#)³⁰¹² - Fix binding const member functions as actions
- [Issue #719](#)³⁰¹³ - Create performance counter exposing compression ratio
- [Issue #718](#)³⁰¹⁴ - Add possibility to compress parcel data
- [Issue #717](#)³⁰¹⁵ - `strip_credit_from_gid` has misleading semantics
- [Issue #716](#)³⁰¹⁶ - Non-option arguments to programs run using `pbsdsh` must be before `--hpx:nodes`, contrary to directions
- [Issue #715](#)³⁰¹⁷ - Re-thrown exceptions should retain the original call site
- [Issue #714](#)³⁰¹⁸ - failed assertion in debug mode
- [Issue #713](#)³⁰¹⁹ - Add performance counters monitoring connection caches
- [Issue #712](#)³⁰²⁰ - Adjust parcel related performance counters to be connection type specific
- [Issue #711](#)³⁰²¹ - configuration failure
- [Issue #710](#)³⁰²² - Error “timed out while trying to find room in the connection cache” when trying to start multiple localities on a single computer
- [Issue #709](#)³⁰²³ - Add new thread state ‘staged’ referring to task descriptions
- [Issue #708](#)³⁰²⁴ - Detect/mitigate bad non-system installs of GCC on Redhat systems
- [Issue #707](#)³⁰²⁵ - Many examples do not link with Git HEAD version
- [Issue #706](#)³⁰²⁶ - `hpx::init` removes portions of non-option command line arguments before last = sign
- [Issue #705](#)³⁰²⁷ - Create rolling average and median aggregating performance counters
- [Issue #704](#)³⁰²⁸ - Create performance counter to expose thread queue waiting time
- [Issue #703](#)³⁰²⁹ - Add support to HPX build system to find `libcrtool.a` and related headers
- [Issue #699](#)³⁰³⁰ - Generalize instrumentation support
- [Issue #698](#)³⁰³¹ - compilation failure with `hwloc` absent
- [Issue #697](#)³⁰³² - Performance counter counts should be zero indexed
- [Issue #696](#)³⁰³³ - Distributed problem

³⁰¹² <https://github.com/STELLAR-GROUP/hpx/issues/722>

³⁰¹³ <https://github.com/STELLAR-GROUP/hpx/issues/719>

³⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/718>

³⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/717>

³⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/716>

³⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/715>

³⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/714>

³⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/713>

³⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/712>

³⁰²¹ <https://github.com/STELLAR-GROUP/hpx/issues/711>

³⁰²² <https://github.com/STELLAR-GROUP/hpx/issues/710>

³⁰²³ <https://github.com/STELLAR-GROUP/hpx/issues/709>

³⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/708>

³⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/707>

³⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/706>

³⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/705>

³⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/704>

³⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/703>

³⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/699>

³⁰³¹ <https://github.com/STELLAR-GROUP/hpx/issues/698>

³⁰³² <https://github.com/STELLAR-GROUP/hpx/issues/697>

³⁰³³ <https://github.com/STELLAR-GROUP/hpx/issues/696>

- Issue #695³⁰³⁴ - Bad perf counter time printed
- Issue #693³⁰³⁵ - `--help` doesn't print component specific command line options
- Issue #692³⁰³⁶ - SLURM support broken
- Issue #691³⁰³⁷ - exception while executing any application linked with hwloc
- Issue #690³⁰³⁸ - `thread_id_test` and `thread_launcher_test` failing
- Issue #689³⁰³⁹ - Make the buildbots use hwloc
- Issue #687³⁰⁴⁰ - compilation error fix (hwloc_topology)
- Issue #686³⁰⁴¹ - Linker Error for Applications
- Issue #684³⁰⁴² - Pinning of service thread fails when number of worker threads equals the number of cores
- Issue #682³⁰⁴³ - Add performance counters exposing number of stolen threads
- Issue #681³⁰⁴⁴ - Add `apply_continue` for asynchronous chaining of actions
- Issue #679³⁰⁴⁵ - Remove obsolete `async_callback` API functions
- Issue #678³⁰⁴⁶ - Add new API for setting/triggering LCOs
- Issue #677³⁰⁴⁷ - Add `async_continue` for true continuation style actions
- Issue #676³⁰⁴⁸ - Buildbot for gcc 4.4 broken
- Issue #675³⁰⁴⁹ - Partial preprocessing broken
- Issue #674³⁰⁵⁰ - HPX segfaults when built with gcc 4.7
- Issue #673³⁰⁵¹ - `use_guard_pages` has inconsistent preprocessor guards
- Issue #672³⁰⁵² - External build breaks if library path has spaces
- Issue #671³⁰⁵³ - release tarballs are tarbombs
- Issue #670³⁰⁵⁴ - CMake won't find Boost headers in layout=versioned install
- Issue #669³⁰⁵⁵ - Links in docs to source files broken if not installed
- Issue #667³⁰⁵⁶ - Not reading ini file properly

³⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/695>

³⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/693>

³⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/692>

³⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/691>

³⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/690>

³⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/689>

³⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/687>

³⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/686>

³⁰⁴² <https://github.com/STELLAR-GROUP/hpx/issues/684>

³⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/682>

³⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/681>

³⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/679>

³⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/678>

³⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/677>

³⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/676>

³⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/675>

³⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/674>

³⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/673>

³⁰⁵² <https://github.com/STELLAR-GROUP/hpx/issues/672>

³⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/671>

³⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/670>

³⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/669>

³⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/667>

- Issue #664³⁰⁵⁷ - Adapt new meanings of ‘const’ and ‘mutable’
- Issue #661³⁰⁵⁸ - Implement BTL Parcel port
- Issue #655³⁰⁵⁹ - Make HPX work with the “decltype” result_of
- Issue #647³⁰⁶⁰ - documentation for specifying the number of high priority threads
--hpx:high-priority-threads
- Issue #643³⁰⁶¹ - Error parsing host file
- Issue #642³⁰⁶² - HWLoc issue with TAU
- Issue #639³⁰⁶³ - Logging potentially suspends a running thread
- Issue #634³⁰⁶⁴ - Improve error reporting from parcel layer
- Issue #627³⁰⁶⁵ - Add tests for async and apply overloads that accept regular C++ functions
- Issue #626³⁰⁶⁶ - hpx/future.hpp header
- Issue #601³⁰⁶⁷ - Intel support
- Issue #557³⁰⁶⁸ - Remove action codes
- Issue #531³⁰⁶⁹ - AGAS request and response classes should use switch statements
- Issue #529³⁰⁷⁰ - Investigate the state of hwloc support
- Issue #526³⁰⁷¹ - Make HPX aware of hyper-threading
- Issue #518³⁰⁷² - Create facilities allowing to use plain arrays as action arguments
- Issue #473³⁰⁷³ - hwloc thread binding is broken on CPUs with hyperthreading
- Issue #383³⁰⁷⁴ - Change result type detection for hpx::util::bind to use result_of protocol
- Issue #341³⁰⁷⁵ - Consolidate route code
- Issue #219³⁰⁷⁶ - Only copy arguments into actions once
- Issue #177³⁰⁷⁷ - Implement distributed AGAS
- Issue #43³⁰⁷⁸ - Support for Darwin (Xcode + Clang)

³⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/664>

³⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/661>

³⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/655>

³⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/647>

³⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/643>

³⁰⁶² <https://github.com/STELLAR-GROUP/hpx/issues/642>

³⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/639>

³⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/634>

³⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/627>

³⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/626>

³⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/601>

³⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/557>

³⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/531>

³⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/529>

³⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/526>

³⁰⁷² <https://github.com/STELLAR-GROUP/hpx/issues/518>

³⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/473>

³⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/383>

³⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/341>

³⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/219>

³⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/177>

³⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/43>

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 improvements. We dedicated much attention to performance and we were able to significantly improve the threading and networking subsystems.

We successfully ported *HPX* to the Android platform. *HPX* applications now not only can run on mobile devices, but we support heterogeneous applications running across architecture boundaries. At the Supercomputing Conference 2012 we demonstrated connecting Android tablets to simulations running on a Linux cluster. The Android tablet was used to query performance counters from the Linux simulation and to steer its parameters.

We successfully ported *HPX* to Mac OSX (using the Clang compiler). Thanks to Pyry Jahkola for contributing the corresponding patches. Please see the section [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](#)³⁰⁷⁹. 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](#)³⁰⁸⁰).

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release. This is by far the longest list of newly implemented features and fixed issues for any of HPX' releases so far.

- [Issue #666](#)³⁰⁸¹ - Segfault on calling `hpx::finalize` twice
- [Issue #665](#)³⁰⁸² - Adding declaration `num_of_cores`
- [Issue #662](#)³⁰⁸³ - `pkgconfig` is building wrong
- [Issue #660](#)³⁰⁸⁴ - Need `uninterrupt` function
- [Issue #659](#)³⁰⁸⁵ - Move our logging library into a different namespace

³⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/>

³⁰⁸⁰ https://www.boost.org/LICENSE_1_0.txt

³⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/666>

³⁰⁸² <https://github.com/STELLAR-GROUP/hpx/issues/665>

³⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/662>

³⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/660>

³⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/659>

- [Issue #658](#)³⁰⁸⁶ - Dynamic performance counter types are broken
- [Issue #657](#)³⁰⁸⁷ - HPX v0.9.5 (RC1) hello_world example segfaulting
- [Issue #656](#)³⁰⁸⁸ - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- [Issue #654](#)³⁰⁸⁹ - Integrate the Boost auto_index tool with documentation
- [Issue #653](#)³⁰⁹⁰ - Make HPX build on OS X + Clang + libc++
- [Issue #651](#)³⁰⁹¹ - Add fine-grained control for thread pinning
- [Issue #650](#)³⁰⁹² - Command line no error message when using -hpx:(anything)
- [Issue #645](#)³⁰⁹³ - Command line aliases don't work in [teletype]“@file“[c++]
- [Issue #644](#)³⁰⁹⁴ - Terminated threads are not always properly cleaned up
- [Issue #640](#)³⁰⁹⁵ - future_data<T>::set_on_completed_ used without locks
- [Issue #638](#)³⁰⁹⁶ - hpx build with intel compilers fails on linux
- [Issue #637](#)³⁰⁹⁷ - -copy-dt-needed-entries breaks with gold
- [Issue #635](#)³⁰⁹⁸ - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- [Issue #633](#)³⁰⁹⁹ - Re-add examples to final 0.9.5 release
- [Issue #632](#)³¹⁰⁰ - Example thread_aware_timer is broken
- [Issue #631](#)³¹⁰¹ - FFT application throws error in parcellayer
- [Issue #630](#)³¹⁰² - Event synchronization example is broken
- [Issue #629](#)³¹⁰³ - Waiting on futures hangs
- [Issue #628](#)³¹⁰⁴ - Add an HPX_ALWAYS_ASSERT macro
- [Issue #625](#)³¹⁰⁵ - Port coroutines context switch benchmark
- [Issue #621](#)³¹⁰⁶ - New INI section for stack sizes
- [Issue #618](#)³¹⁰⁷ - pkg_config support does not work with a HPX debug build
- [Issue #617](#)³¹⁰⁸ - hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: 'get_thread_id' was not declared in this scope

³⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/658>

³⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/657>

³⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/656>

³⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/654>

³⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/653>

³⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/651>

³⁰⁹² <https://github.com/STELLAR-GROUP/hpx/issues/650>

³⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/645>

³⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/644>

³⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/640>

³⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/638>

³⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/637>

³⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/635>

³⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/633>

³¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/632>

³¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/631>

³¹⁰² <https://github.com/STELLAR-GROUP/hpx/issues/630>

³¹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/629>

³¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/628>

³¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/625>

³¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/621>

³¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/618>

³¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/617>

- Issue #616³¹⁰⁹ - Change wait_xxx not to use locking
- Issue #615³¹¹⁰ - Revert visibility 'fix' (fb0b6b8245dad1127b0c25ebafd9386b3945cca9)
- Issue #614³¹¹¹ - Fix Dataflow linker error
- Issue #613³¹¹² - find_here should throw an exception on failure
- Issue #612³¹¹³ - Thread phase doesn't show up in debug mode
- Issue #611³¹¹⁴ - Make stack guard pages configurable at runtime (initialization time)
- Issue #610³¹¹⁵ - Co-Locate Components
- Issue #609³¹¹⁶ - future_overhead
- Issue #608³¹¹⁷ - --hpx:list-counter-infos problem
- Issue #607³¹¹⁸ - Update Boost.Context based backend for coroutines
- Issue #606³¹¹⁹ - 1d_wave_equation is not working
- Issue #605³¹²⁰ - Any C++ function that has serializable arguments and a serializable return type should be remotable
- Issue #604³¹²¹ - Connecting localities isn't working anymore
- Issue #603³¹²² - Do not verify any ini entries read from a file
- Issue #602³¹²³ - Rename argument_size to type_size/ added implementation to get parcel size
- Issue #599³¹²⁴ - Enable locality specific command line options
- Issue #598³¹²⁵ - Need an API that accesses the performance counter reporting the system uptime
- Issue #597³¹²⁶ - compiling on ranger
- Issue #595³¹²⁷ - I need a place to store data in a thread self pointer
- Issue #594³¹²⁸ - 32/64 interoperability
- Issue #593³¹²⁹ - Warn if logging is disabled at compile time but requested at runtime
- Issue #592³¹³⁰ - Add optional argument value to --hpx:list-counters and --hpx:list-counter-infos

³¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/616>

³¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/615>

³¹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/614>

³¹¹² <https://github.com/STELLAR-GROUP/hpx/issues/613>

³¹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/612>

³¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/611>

³¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/610>

³¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/609>

³¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/608>

³¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/607>

³¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/606>

³¹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/605>

³¹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/604>

³¹²² <https://github.com/STELLAR-GROUP/hpx/issues/603>

³¹²³ <https://github.com/STELLAR-GROUP/hpx/issues/602>

³¹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/599>

³¹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/598>

³¹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/597>

³¹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/595>

³¹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/594>

³¹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/593>

³¹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/592>

- [Issue #591](#)³¹³¹ - Allow for wildcards in performance counter names specified with `--hpx:print-counter`
- [Issue #590](#)³¹³² - Local promise semantic differences
- [Issue #589](#)³¹³³ - Create API to query performance counter names
- [Issue #587](#)³¹³⁴ - Add `get_num_localities` and `get_num_threads` to AGAS API
- [Issue #586](#)³¹³⁵ - Adjust local AGAS cache size based on number of localities
- [Issue #585](#)³¹³⁶ - Error while using counters in HPX
- [Issue #584](#)³¹³⁷ - counting argument size of actions, initial pass.
- [Issue #581](#)³¹³⁸ - Remove `RemoteResult` template parameter for `future<>`
- [Issue #580](#)³¹³⁹ - Add possibility to hook into actions
- [Issue #578](#)³¹⁴⁰ - Use angle brackets in HPX error dumps
- [Issue #576](#)³¹⁴¹ - Exception incorrectly thrown when `--help` is used
- [Issue #575](#)³¹⁴² - `HPX(bad_component_type)` with gcc 4.7.2 and boost 1.51
- [Issue #574](#)³¹⁴³ - `--hpx:connect` command line parameter not working correctly
- [Issue #571](#)³¹⁴⁴ - `hpx::wait()` (callback version) should pass the future to the callback function
- [Issue #570](#)³¹⁴⁵ - `hpx::wait` should operate on `boost::arrays` and `std::lists`
- [Issue #569](#)³¹⁴⁶ - Add a logging sink for Android
- [Issue #568](#)³¹⁴⁷ - 2-argument version of `HPX_DEFINE_COMPONENT_ACTION`
- [Issue #567](#)³¹⁴⁸ - Connecting to a running HPX application works only once
- [Issue #565](#)³¹⁴⁹ - HPX doesn't shutdown properly
- [Issue #564](#)³¹⁵⁰ - Partial preprocessing of new component creation interface
- [Issue #563](#)³¹⁵¹ - Add `hpx::start/hpx::stop` to avoid blocking main thread
- [Issue #562](#)³¹⁵² - All command line arguments swallowed by `hpx`
- [Issue #561](#)³¹⁵³ - `Boost.Tuple` is not move aware

³¹³¹ <https://github.com/STELLAR-GROUP/hpx/issues/591>

³¹³² <https://github.com/STELLAR-GROUP/hpx/issues/590>

³¹³³ <https://github.com/STELLAR-GROUP/hpx/issues/589>

³¹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/587>

³¹³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/586>

³¹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/585>

³¹³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/584>

³¹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/581>

³¹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/580>

³¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/578>

³¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/576>

³¹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/575>

³¹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/574>

³¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/571>

³¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/570>

³¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/569>

³¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/568>

³¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/567>

³¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/565>

³¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/564>

³¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/563>

³¹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/562>

³¹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/561>

- Issue #558³¹⁵⁴ - `boost::shared_ptr<>` style semantics/syntax for client classes
- Issue #556³¹⁵⁵ - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
- Issue #555³¹⁵⁶ - `BOOST_FORCEINLINE` does not name a type
- Issue #554³¹⁵⁷ - Possible race condition in `thread::get_id()`
- Issue #552³¹⁵⁸ - Move `enable_client_base`
- Issue #550³¹⁵⁹ - Add stack size category 'huge'
- Issue #549³¹⁶⁰ - ShenEOS run seg-faults on single or distributed runs
- Issue #545³¹⁶¹ - `AUTOGLOB` broken for `add_hpx_component`
- Issue #542³¹⁶² - `FindHPX_HDF5` still searches multiple times
- Issue #541³¹⁶³ - Quotes around application name in `hpx::init`
- Issue #539³¹⁶⁴ - Race condition occurring with new lightweight threads
- Issue #535³¹⁶⁵ - `hpx_run_tests.py` exits with no error code when tests are missing
- Issue #530³¹⁶⁶ - Thread description(<unknown>) in logs
- Issue #523³¹⁶⁷ - Make thread objects more lightweight
- Issue #521³¹⁶⁸ - `hpx::error_code` is not usable for lightweight error handling
- Issue #520³¹⁶⁹ - Add full user environment to HPX logs
- Issue #519³¹⁷⁰ - Build succeeds, running fails
- Issue #517³¹⁷¹ - Add a guard page to linux coroutine stacks
- Issue #516³¹⁷² - `hpx::thread::detach` suspends while holding locks, leads to hang in debug
- Issue #514³¹⁷³ - Preprocessed headers for `<hpx/apply.hpp>` don't compile
- Issue #513³¹⁷⁴ - Buildbot configuration problem
- Issue #512³¹⁷⁵ - Implement action based stack size customization
- Issue #511³¹⁷⁶ - Move action priority into a separate type trait

³¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/558>

³¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/556>

³¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/555>

³¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/554>

³¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/552>

³¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/550>

³¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/549>

³¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/545>

³¹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/542>

³¹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/541>

³¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/539>

³¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/535>

³¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/530>

³¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/523>

³¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/521>

³¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/520>

³¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/519>

³¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/517>

³¹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/516>

³¹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/514>

³¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/513>

³¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/512>

³¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/511>

- Issue #510³¹⁷⁷ - trunk broken
- Issue #507³¹⁷⁸ - no matching function for call to `boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(h`
- Issue #505³¹⁷⁹ - `undefined_symbol` regression test currently failing
- Issue #502³¹⁸⁰ - Adding OpenCL and OCLM support to HPX for Windows and Linux
- Issue #501³¹⁸¹ - `find_package(HPX)` sets cmake output variables
- Issue #500³¹⁸² - `wait_any/wait_all` are badly named
- Issue #499³¹⁸³ - Add support for disabling pbs support in pbs runs
- Issue #498³¹⁸⁴ - Error during no-cache runs
- Issue #496³¹⁸⁵ - Add partial preprocessing support to cmake
- Issue #495³¹⁸⁶ - Support HPX modules exporting startup/shutdown functions only
- Issue #494³¹⁸⁷ - Allow modules to specify when to run startup/shutdown functions
- Issue #493³¹⁸⁸ - Avoid constructing a string in `make_success_code`
- Issue #492³¹⁸⁹ - Performance counter creation is no longer synchronized at startup
- Issue #491³¹⁹⁰ - Performance counter creation is no longer synchronized at startup
- Issue #490³¹⁹¹ - Sheneos on `_completed_bulk` seg fault in distributed
- Issue #489³¹⁹² - compiling issue with g++44
- Issue #488³¹⁹³ - Adding OpenCL and OCLM support to HPX for the MSVC platform
- Issue #487³¹⁹⁴ - FindHPX.cmake problems
- Issue #485³¹⁹⁵ - Change `distributing_factory` and `binpacking_factory` to use bulk creation
- Issue #484³¹⁹⁶ - Change `HPX_DONT_USE_PREPROCESSED_FILES` to `HPX_USE_PREPROCESSED_FILES`
- Issue #483³¹⁹⁷ - Memory counter for Windows
- Issue #479³¹⁹⁸ - strange errors appear when requesting performance counters on multiple nodes
- Issue #477³¹⁹⁹ - Create (global) timer for multi-threaded measurements

³¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/510>

³¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/507>

³¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/505>

³¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/502>

³¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/501>

³¹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/500>

³¹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/499>

³¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/498>

³¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/496>

³¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/495>

³¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/494>

³¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/493>

³¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/492>

³¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/491>

³¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/490>

³¹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/489>

³¹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/488>

³¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/487>

³¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/485>

³¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/484>

³¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/483>

³¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/479>

³¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/477>

- Issue #472³²⁰⁰ - Add partial preprocessing using Wave
- Issue #471³²⁰¹ - Segfault stack traces don't show up in release
- Issue #468³²⁰² - External projects need to link with internal components
- Issue #462³²⁰³ - Startup/shutdown functions are called more than once
- Issue #458³²⁰⁴ - Consolidate `hpx::util::high_resolution_timer` and `hpx::util::high_resolution_clock`
- Issue #457³²⁰⁵ - index out of bounds in `allgather_and_gate` on 4 cores or more
- Issue #448³²⁰⁶ - Make HPX compile with clang
- Issue #447³²⁰⁷ - 'make tests' should execute tests on local installation
- Issue #446³²⁰⁸ - Remove SVN-related code from the codebase
- Issue #444³²⁰⁹ - race condition in `smp`
- Issue #441³²¹⁰ - Patched Boost.Serialization headers should only be installed if needed
- Issue #439³²¹¹ - Components using `HPX_REGISTER_STARTUP_MODULE` fail to compile with MSVC
- Issue #436³²¹² - Verify that no locks are being held while threads are suspended
- Issue #435³²¹³ - Installing HPX should not clobber existing Boost installation
- Issue #434³²¹⁴ - Logging external component failed (Boost 1.50)
- Issue #433³²¹⁵ - Runtime crash when building all examples
- Issue #432³²¹⁶ - Dataflow hangs on 512 cores/64 nodes
- Issue #430³²¹⁷ - Problem with distributing factory
- Issue #424³²¹⁸ - File paths referring to XSL-files need to be properly escaped
- Issue #417³²¹⁹ - Make dataflow LCOs work out of the box by using partial preprocessing
- Issue #413³²²⁰ - `hpx_svnversion.py` fails on Windows
- Issue #412³²²¹ - Make `hpx::error_code` equivalent to `hpx::exception`
- Issue #398³²²² - HPX clobbers out-of-tree application specific CMake variables (specifically `CMAKE_BUILD_TYPE`)

³²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/472>

³²⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/471>

³²⁰² <https://github.com/STELLAR-GROUP/hpx/issues/468>

³²⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/462>

³²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/458>

³²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/457>

³²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/448>

³²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/447>

³²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/446>

³²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/444>

³²¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/441>

³²¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/439>

³²¹² <https://github.com/STELLAR-GROUP/hpx/issues/436>

³²¹³ <https://github.com/STELLAR-GROUP/hpx/issues/435>

³²¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/434>

³²¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/433>

³²¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/432>

³²¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/430>

³²¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/424>

³²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/417>

³²²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/413>

³²²¹ <https://github.com/STELLAR-GROUP/hpx/issues/412>

³²²² <https://github.com/STELLAR-GROUP/hpx/issues/398>

- [Issue #394](#)³²²³ - Remove code generating random port numbers for network
- [Issue #378](#)³²²⁴ - ShenEOS scaling issues
- [Issue #354](#)³²²⁵ - Create a coroutines wrapper for Boost.Context
- [Issue #349](#)³²²⁶ - Commandline option `--localities=N/-lN` should be necessary only on AGAS locality
- [Issue #334](#)³²²⁷ - Add `auto_index` support to cmake based documentation toolchain
- [Issue #318](#)³²²⁸ - Network benchmarks
- [Issue #317](#)³²²⁹ - Implement network performance counters
- [Issue #310](#)³²³⁰ - Duplicate logging entries
- [Issue #230](#)³²³¹ - Add compile time option to disable thread debugging info
- [Issue #171](#)³²³² - Add an INI option to turn off deadlock detection independently of logging
- [Issue #170](#)³²³³ - OSHL internal counters are incorrect
- [Issue #103](#)³²³⁴ - Better diagnostics for multiple component/action registrations under the same name
- [Issue #48](#)³²³⁵ - Support for Darwin (Xcode + Clang)
- [Issue #21](#)³²³⁶ - 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.).

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](#)³²³⁷.
- *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.

³²²³ <https://github.com/STELLAR-GROUP/hpx/issues/394>

³²²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/378>

³²²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/354>

³²²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/349>

³²²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/334>

³²²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/318>

³²²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/317>

³²³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/310>

³²³¹ <https://github.com/STELLAR-GROUP/hpx/issues/230>

³²³² <https://github.com/STELLAR-GROUP/hpx/issues/171>

³²³³ <https://github.com/STELLAR-GROUP/hpx/issues/170>

³²³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/103>

³²³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/48>

³²³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/21>

³²³⁷ <http://www.open-std.org/jtc1/sc22/wg21>

- 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 its 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();
}
```

This removes the immediate dependency on the [Boost.Program Options](https://www.boost.org/doc/html/program_options.html)³²³⁸ 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](https://www.boost.org/doc/html/program_options.html)³²³⁹, the minimal program can be written as:

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

(continues on next page)

³²³⁸ https://www.boost.org/doc/html/program_options.html

³²³⁹ https://www.boost.org/doc/html/program_options.html

(continued from previous page)

```
// 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 #71³²⁴⁰ - GIDs that are not serialized via `handle_gid<>` should raise an exception
- Issue #105³²⁴¹ - Allow for `hpx::util::functions` to be registered in the AGAS symbolic namespace
- Issue #107³²⁴² - Nasty threadmanger race condition (reproducible in `sheneos_test`)
- Issue #108³²⁴³ - Add millisecond resolution to *HPX* logs on Linux
- Issue #110³²⁴⁴ - Shutdown hang in distributed with release build
- Issue #116³²⁴⁵ - Don't use TSS for the applier and runtime pointers
- Issue #162³²⁴⁶ - Move local synchronous execution shortcut from `hpx::function` to the applier
- Issue #172³²⁴⁷ - Cache sources in CMake and check if they change manually
- Issue #178³²⁴⁸ - Add an INI option to turn off ranged-based AGAS caching

³²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/71>

³²⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/105>

³²⁴² <https://github.com/STELLAR-GROUP/hpx/issues/107>

³²⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/108>

³²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/110>

³²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/116>

³²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/162>

³²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/172>

³²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/178>

- Issue #187³²⁴⁹ - Support for disabling performance counter deployment
- Issue #202³²⁵⁰ - Support for sending performance counter data to a specific file
- Issue #218³²⁵¹ - boost.coroutines allows different stack sizes, but stack pool is unaware of this
- Issue #231³²⁵² - Implement movable `boost::bind`
- Issue #232³²⁵³ - Implement movable `boost::function`
- Issue #236³²⁵⁴ - Allow binding `hpx::util::function` to actions
- Issue #239³²⁵⁵ - Replace `hpx::function` with `hpx::util::function`
- Issue #240³²⁵⁶ - Can't specify `RemoteResult` with `lcos::async`
- Issue #242³²⁵⁷ - REGISTER_TEMPLATE support for plain actions
- Issue #243³²⁵⁸ - `handle_gid<>` support for `hpx::util::function`
- Issue #245³²⁵⁹ - `*_c_cache` code throws an exception if the queried GID is not in the local cache
- Issue #246³²⁶⁰ - Undefined references in dataflow/adaptive1d example
- Issue #252³²⁶¹ - Problems configuring sheneos with CMake
- Issue #254³²⁶² - Lifetime of components doesn't end when client goes out of scope
- Issue #259³²⁶³ - CMake does not detect that MSVC10 has lambdas
- Issue #260³²⁶⁴ - `io_service_pool` segfault
- Issue #261³²⁶⁵ - Late parcel executed outside of `pxthread`
- Issue #263³²⁶⁶ - Cannot select allocator with CMake
- Issue #264³²⁶⁷ - Fix allocator select
- Issue #267³²⁶⁸ - Runtime error for `hello_world`
- Issue #269³²⁶⁹ - `pthread_affinity_np` test fails to compile
- Issue #270³²⁷⁰ - Compiler noise due to `-Wcast-qual`
- Issue #275³²⁷¹ - Problem with configuration tests/include paths on Gentoo

³²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/187>

³²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/202>

³²⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/218>

³²⁵² <https://github.com/STELLAR-GROUP/hpx/issues/231>

³²⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/232>

³²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/236>

³²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/239>

³²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/240>

³²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/242>

³²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/243>

³²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/245>

³²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/246>

³²⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/252>

³²⁶² <https://github.com/STELLAR-GROUP/hpx/issues/254>

³²⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/259>

³²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/260>

³²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/261>

³²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/263>

³²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/264>

³²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/267>

³²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/269>

³²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/270>

³²⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/275>

- Issue #325³²⁷² - Sheneos is 200-400 times slower than the fortran equivalent
- Issue #331³²⁷³ - `hpx::init` and `hpx_main()` should not depend on `program_options`
- Issue #333³²⁷⁴ - Add doxygen support to CMake for doc toolchain
- Issue #340³²⁷⁵ - Performance counters for parcels
- Issue #346³²⁷⁶ - Component loading error when running `hello_world` in distributed on MSVC2010
- Issue #362³²⁷⁷ - Missing initializer error
- Issue #363³²⁷⁸ - Parcel port serialization error
- Issue #366³²⁷⁹ - Parcel buffering leads to types incompatible exception
- Issue #368³²⁸⁰ - Scalable alternative to `rand()` needed for *HPX*
- Issue #369³²⁸¹ - IB over IP is substantially slower than just using standard TCP/IP
- Issue #374³²⁸² - `hpx::lcos::wait` should work with dataflows and arbitrary classes meeting the future interface
- Issue #375³²⁸³ - Conflicting/ambiguous overloads of `hpx::lcos::wait`
- Issue #376³²⁸⁴ - `Find_HPX.cmake` should set CMake variable `HPX_FOUND` for out of tree builds
- Issue #377³²⁸⁵ - ShenEOS interpolate bulk and `interpolate_one_bulk` are broken
- Issue #379³²⁸⁶ - Add support for distributed runs under SLURM
- Issue #382³²⁸⁷ - `_Unwind_Word` not declared in `boost.backtrace`
- Issue #387³²⁸⁸ - Doxygen should look only at list of specified files
- Issue #388³²⁸⁹ - Running `make install` on an out-of-tree application is broken
- Issue #391³²⁹⁰ - Out-of-tree application segfaults when running in `qsub`
- Issue #392³²⁹¹ - Remove `HPX_NO_INSTALL` option from cmake build system
- Issue #396³²⁹² - Pragma related warnings when compiling with older gcc versions
- Issue #399³²⁹³ - Out of tree component build problems
- Issue #400³²⁹⁴ - Out of source builds on Windows: linker should not receive compiler flags

³²⁷² <https://github.com/STELLAR-GROUP/hpx/issues/325>

³²⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/331>

³²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/333>

³²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/340>

³²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/346>

³²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/362>

³²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/363>

³²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/366>

³²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/368>

³²⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/369>

³²⁸² <https://github.com/STELLAR-GROUP/hpx/issues/374>

³²⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/375>

³²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/376>

³²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/377>

³²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/379>

³²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/382>

³²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/387>

³²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/388>

³²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/391>

³²⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/392>

³²⁹² <https://github.com/STELLAR-GROUP/hpx/issues/396>

³²⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/399>

³²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/400>

- [Issue #401](#)³²⁹⁵ - Out of source builds on Windows: components need to be linked with `hpx_serialization`
- [Issue #404](#)³²⁹⁶ - gfortran fails to link automatically when fortran files are present
- [Issue #405](#)³²⁹⁷ - Inability to specify linking order for external libraries
- [Issue #406](#)³²⁹⁸ - Adapt action limits such that dataflow applications work without additional defines
- [Issue #415](#)³²⁹⁹ - `locality_results` is not a member of `hpx::components::server`
- [Issue #425](#)³³⁰⁰ - Breaking changes to `traits::*result wrt std::vector<id_type>`
- [Issue #426](#)³³⁰¹ - 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](#)³³⁰² - Don't require install path to be known at compile time.
- [Issue #371](#)³³⁰³ - Add `hpx iostreams` to standard build.
- [Issue #384](#)³³⁰⁴ - Fix compilation with GCC 4.7.
- [Issue #390](#)³³⁰⁵ - Remove `keep_factory_alive` startup call from ShenEOS; add shutdown call to `H5close`.
- [Issue #393](#)³³⁰⁶ - Thread affinity control is broken.

Bug fixes (commits)

Here is a list of the important commits included in this point release:

- `r7642` - External: Fix backtrace memory violation.
- **`r7775` - Components: Fix symbol visibility bug with component startup** providers. This prevents one components providers from overriding another components.
- `r7778` - Components: Fix startup/shutdown provider shadowing issues.

³²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/401>

³²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/404>

³²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/405>

³²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/406>

³²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/415>

³³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/425>

³³⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/426>

³³⁰² <https://github.com/STELLAR-GROUP/hpx/issues/295>

³³⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/371>

³³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/384>

³³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/390>

³³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/393>

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](#)³³⁰⁷ (thanks to Matthew Anderson for contributing this application).
- [GTC \(Gyrokinetic Toroidal Code\)](#)³³⁰⁸: a skeleton for particle in cell type codes.
- Random Memory Access: an example demonstrating random memory accesses in a large array
- [ShenEOS example](#)³³⁰⁹, 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`.
- Restructured the Futures sub-system, renaming types in accordance with the terminology used by the C++11 ISO standard.

³³⁰⁷ <http://www.graph500.org/>

³³⁰⁸ <http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/gtc/>

³³⁰⁹ <http://stellarcollapse.org/equationofstate>

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #31³³¹⁰ - Specialize `handle_gid<>` for examples and tests
- Issue #72³³¹¹ - Fix AGAS reference counting
- Issue #104³³¹² - heartbeat throws an exception when decrefing the performance counter it's watching
- Issue #111³³¹³ - throttle causes an exception on the target application
- Issue #142³³¹⁴ - One failed component loading causes an unrelated component to fail
- Issue #165³³¹⁵ - Remote exception propagation bug in AGAS reference counting test
- Issue #186³³¹⁶ - Test credit exhaustion/splitting (e.g. `prepare_gid` and symbol NS)
- Issue #188³³¹⁷ - Implement remaining AGAS reference counting test cases
- Issue #258³³¹⁸ - No type checking of GIDs in stubs classes
- Issue #271³³¹⁹ - Seg fault/shared pointer assertion in distributed code
- Issue #281³³²⁰ - CMake options need descriptive text
- Issue #283³³²¹ - AGAS caching broken (`gva_cache` needs to be rewritten with ICL)
- Issue #285³³²² - `HPX_INSTALL` root directory not the same as `CMAKE_INSTALL_PREFIX`
- Issue #286³³²³ - New segfault in dataflow applications
- Issue #289³³²⁴ - Exceptions should only be logged if not handled
- Issue #290³³²⁵ - c++11 tests failure
- Issue #293³³²⁶ - Build target for component libraries
- Issue #296³³²⁷ - Compilation error with Boost V1.49rc1
- Issue #298³³²⁸ - Illegal instructions on termination
- Issue #299³³²⁹ - gravity aborts with multiple threads
- Issue #301³³³⁰ - Build error with Boost trunk

³³¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/31>

³³¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/72>

³³¹² <https://github.com/STELLAR-GROUP/hpx/issues/104>

³³¹³ <https://github.com/STELLAR-GROUP/hpx/issues/111>

³³¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/142>

³³¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/165>

³³¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/186>

³³¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/188>

³³¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/258>

³³¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/271>

³³²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/281>

³³²¹ <https://github.com/STELLAR-GROUP/hpx/issues/283>

³³²² <https://github.com/STELLAR-GROUP/hpx/issues/285>

³³²³ <https://github.com/STELLAR-GROUP/hpx/issues/286>

³³²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/289>

³³²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/290>

³³²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/293>

³³²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/296>

³³²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/298>

³³²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/299>

³³³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/301>

- Issue #303³³³¹ - Logging assertion failure in distributed runs
- Issue #304³³³² - Exception ‘what’ strings are lost when exceptions from `decode_parcel` are reported
- Issue #306³³³³ - Performance counter user interface issues
- Issue #307³³³⁴ - Logging exception in distributed runs
- Issue #308³³³⁵ - Logging deadlocks in distributed
- Issue #309³³³⁶ - Reference counting test failures and exceptions
- Issue #311³³³⁷ - Merge AGAS `remote_interface` with the `runtime_support` object
- Issue #314³³³⁸ - Object tracking for `id_types`
- Issue #315³³³⁹ - Remove `handle_gid` and handle credit splitting in `id_type` serialization
- Issue #320³³⁴⁰ - `applier::get_locality_id()` should return an error value (or throw an exception)
- Issue #321³³⁴¹ - Optimization for `id_types` which are never split should be restored
- Issue #322³³⁴² - Command line processing ignored with Boost 1.47.0
- Issue #323³³⁴³ - Credit exhaustion causes object to stay alive
- Issue #324³³⁴⁴ - Duplicate exception messages
- Issue #326³³⁴⁵ - Integrate Quickbook with CMake
- Issue #329³³⁴⁶ - `--help` and `--version` should still work
- Issue #330³³⁴⁷ - Create `pkg-config` files
- Issue #337³³⁴⁸ - Improve usability of performance counter timestamps
- Issue #338³³⁴⁹ - Non-std exceptions deriving from `std::exceptions` in `tfunc` may be sliced
- Issue #339³³⁵⁰ - Decrease the number of `send_pending_parcel` threads
- Issue #343³³⁵¹ - Dynamically setting the stack size doesn’t work
- Issue #351³³⁵² - ‘make install’ does not update documents
- Issue #353³³⁵³ - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs

³³³¹ <https://github.com/STELLAR-GROUP/hpx/issues/303>

³³³² <https://github.com/STELLAR-GROUP/hpx/issues/304>

³³³³ <https://github.com/STELLAR-GROUP/hpx/issues/306>

³³³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/307>

³³³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/308>

³³³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/309>

³³³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/311>

³³³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/314>

³³³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/315>

³³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/320>

³³⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/321>

³³⁴² <https://github.com/STELLAR-GROUP/hpx/issues/322>

³³⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/323>

³³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/324>

³³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/326>

³³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/329>

³³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/330>

³³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/337>

³³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/338>

³³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/339>

³³⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/343>

³³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/351>

³³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/353>

- [Issue #355](#)³³⁵⁴ - ‘make’ doesn’t do anything after correct configuration
- [Issue #356](#)³³⁵⁵ - Don’t use `hpx::util::static_` in topology code
- [Issue #359](#)³³⁵⁶ - Infinite recursion in `hpx::tuple` serialization
- [Issue #361](#)³³⁵⁷ - Add compile time option to disable logging completely
- [Issue #364](#)³³⁵⁸ - 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.
- `accumulator`, simple component example.

³³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/355>

³³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/356>

³³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/359>

³³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/361>

³³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/364>

- `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 #28³³⁵⁹ - Integrate Windows/Linux CMake code for *HPX* core
- Issue #32³³⁶⁰ - `hpx::cout()` should be `hpx::cout`
- Issue #33³³⁶¹ - AGAS V2 legacy client does not properly handle `error_code`
- Issue #60³³⁶² - AGAS: allow for `registerid` to optionally take ownership of the `gid`
- Issue #62³³⁶³ - `adaptive1d` compilation failure in Fusion
- Issue #64³³⁶⁴ - Parcel subsystem doesn't resolve domain names
- Issue #83³³⁶⁵ - No error handling if no console is available
- Issue #84³³⁶⁶ - No error handling if a hosted locality is treated as the bootstrap server
- Issue #90³³⁶⁷ - Add general commandline option `-N`
- Issue #91³³⁶⁸ - Add possibility to read command line arguments from file
- Issue #92³³⁶⁹ - Always log exceptions/errors to the log file

³³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/28>

³³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/32>

³³⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/33>

³³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/60>

³³⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/62>

³³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/64>

³³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/83>

³³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/84>

³³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/90>

³³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/91>

³³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/92>

- Issue #93³³⁷⁰ - Log the command line/program name
- Issue #95³³⁷¹ - Support for distributed launches
- Issue #97³³⁷² - Attempt to create a bad component type in AMR examples
- Issue #100³³⁷³ - factorial and factorial_get examples trigger AGAS component type assertions
- Issue #101³³⁷⁴ - Segfault when `hpx::process::here()` is called in `fibonacci2`
- Issue #102³³⁷⁵ - `unknown_component_address` in `int_object_semaphore_client`
- Issue #114³³⁷⁶ - `marduk` raises assertion with default parameters
- Issue #115³³⁷⁷ - Logging messages for SMP runs (on the console) shouldn't be buffered
- Issue #119³³⁷⁸ - `marduk` linking strategy breaks other applications
- Issue #121³³⁷⁹ - `pbsdsh` problem
- Issue #123³³⁸⁰ - `marduk`, `dataflow` and `adaptive1d` fail to build
- Issue #124³³⁸¹ - Lower default preprocessing arity
- Issue #125³³⁸² - Move `hpx::detail::diagnostic_information` out of the `detail` namespace
- Issue #126³³⁸³ - Test definitions for AGAS reference counting
- Issue #128³³⁸⁴ - Add averaging performance counter
- Issue #129³³⁸⁵ - Error with `endian.hpp` while building `adaptive1d`
- Issue #130³³⁸⁶ - Bad initialization of performance counters
- Issue #131³³⁸⁷ - Add global startup/shutdown functions to component modules
- Issue #132³³⁸⁸ - Avoid using `auto_ptr`
- Issue #133³³⁸⁹ - On Windows `hpx.dll` doesn't get installed
- Issue #134³³⁹⁰ - `HPX_LIBRARY` does not reflect real library name (on Windows)
- Issue #135³³⁹¹ - Add detection of `unique_ptr` to build system
- Issue #137³³⁹² - Add command line option allowing to repeatedly evaluate performance counters

³³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/93>

³³⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/95>

³³⁷² <https://github.com/STELLAR-GROUP/hpx/issues/97>

³³⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/100>

³³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/101>

³³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/102>

³³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/114>

³³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/115>

³³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/119>

³³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/121>

³³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/123>

³³⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/124>

³³⁸² <https://github.com/STELLAR-GROUP/hpx/issues/125>

³³⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/126>

³³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/128>

³³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/129>

³³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/130>

³³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/131>

³³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/132>

³³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/133>

³³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/134>

³³⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/135>

³³⁹² <https://github.com/STELLAR-GROUP/hpx/issues/137>

- Issue #139³³⁹³ - Logging is broken
- Issue #140³³⁹⁴ - CMake problem on windows
- Issue #141³³⁹⁵ - Move all non-component libraries into \$PREFIX/lib/hpx
- Issue #143³³⁹⁶ - adaptive1d throws an exception with the default command line options
- Issue #146³³⁹⁷ - Early exception handling is broken
- Issue #147³³⁹⁸ - Sheneos doesn't link on Linux
- Issue #149³³⁹⁹ - sheneos_test hangs
- Issue #154³⁴⁰⁰ - Compilation fails for r5661
- Issue #155³⁴⁰¹ - Sine performance counters example chokes on chrono headers
- Issue #156³⁴⁰² - Add build type to --version
- Issue #157³⁴⁰³ - Extend AGAS caching to store gid ranges
- Issue #158³⁴⁰⁴ - r5691 doesn't compile
- Issue #160³⁴⁰⁵ - Re-add AGAS function for resolving a locality to its prefix
- Issue #168³⁴⁰⁶ - Managed components should be able to access their own GID
- Issue #169³⁴⁰⁷ - Rewrite AGAS future pool
- Issue #179³⁴⁰⁸ - Complete switch to request class for AGAS server interface
- Issue #182³⁴⁰⁹ - Sine performance counter is loaded by other examples
- Issue #185³⁴¹⁰ - Write tests for symbol namespace reference counting
- Issue #191³⁴¹¹ - Assignment of read-only variable in point_geometry
- Issue #200³⁴¹² - Seg faults when querying performance counters
- Issue #204³⁴¹³ - --ifnames and suffix stripping needs to be more generic
- Issue #205³⁴¹⁴ - --list-* and --print-counter-* options do not work together and produce no warning
- Issue #207³⁴¹⁵ - Implement decrement entry merging

³³⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/139>

³³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/140>

³³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/141>

³³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/143>

³³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/146>

³³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/147>

³³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/149>

³⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/154>

³⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/155>

³⁴⁰² <https://github.com/STELLAR-GROUP/hpx/issues/156>

³⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/157>

³⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/158>

³⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/160>

³⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/168>

³⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/169>

³⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/179>

³⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/182>

³⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/185>

³⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/191>

³⁴¹² <https://github.com/STELLAR-GROUP/hpx/issues/200>

³⁴¹³ <https://github.com/STELLAR-GROUP/hpx/issues/204>

³⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/205>

³⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/207>

- [Issue #208](#)³⁴¹⁶ - Replace the spinlocks in AGAS with `hpx::lcos::local_mutexes`
- [Issue #210](#)³⁴¹⁷ - Add an `-ifprefix` option
- [Issue #214](#)³⁴¹⁸ - Performance test for PX-thread creation
- [Issue #216](#)³⁴¹⁹ - VS2010 compilation
- [Issue #222](#)³⁴²⁰ - `r6045 context_linux_x86.hpp`
- [Issue #223](#)³⁴²¹ - fibonacci hangs when changing the state of an active thread
- [Issue #225](#)³⁴²² - Active threads end up in the FEB wait queue
- [Issue #226](#)³⁴²³ - VS Build Error for Accumulator Client
- [Issue #228](#)³⁴²⁴ - Move all traits into namespace `hpx::traits`
- [Issue #229](#)³⁴²⁵ - Invalid initialization of reference in `thread_init_data`
- [Issue #235](#)³⁴²⁶ - Invalid GID in `iostreams`
- [Issue #238](#)³⁴²⁷ - Demangle type names for the default implementation of `get_action_name`
- [Issue #241](#)³⁴²⁸ - C++11 support breaks GCC 4.5
- [Issue #247](#)³⁴²⁹ - Reference to temporary with GCC 4.4
- [Issue #248](#)³⁴³⁰ - Seg fault at shutdown with GCC 4.4
- [Issue #253](#)³⁴³¹ - Default component action registration kills compiler
- [Issue #272](#)³⁴³² - G++ unrecognized command line option
- [Issue #273](#)³⁴³³ - quicksort example doesn't compile
- [Issue #277](#)³⁴³⁴ - Invalid CMake logic for Windows

2.11 About HPX

2.11.1 History

The development of High Performance ParallelX (*HPX*) began in 2007. At that time, Hartmut Kaiser became interested in the work done by the ParallelX group at the [Center for Computation and Technology \(CCT\)](#)³⁴³⁵, a multi-disciplinary

³⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/208>

³⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/210>

³⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/214>

³⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/216>

³⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/222>

³⁴²¹ <https://github.com/STELLAR-GROUP/hpx/issues/223>

³⁴²² <https://github.com/STELLAR-GROUP/hpx/issues/225>

³⁴²³ <https://github.com/STELLAR-GROUP/hpx/issues/226>

³⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/228>

³⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/229>

³⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/235>

³⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/238>

³⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/241>

³⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/247>

³⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/248>

³⁴³¹ <https://github.com/STELLAR-GROUP/hpx/issues/253>

³⁴³² <https://github.com/STELLAR-GROUP/hpx/issues/272>

³⁴³³ <https://github.com/STELLAR-GROUP/hpx/issues/273>

³⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/277>

³⁴³⁵ <https://www.cct.lsu.edu>

research institute at [Louisiana State University \(LSU\)](https://www.lsu.edu)³⁴³⁶. 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 [STEIIAR](https://stellar-group.org)³⁴³⁷ group (Systems Technology, Emergent Parallelism, and Algorithm Research). Over time, the team grew to include researchers around the country and the world. In 2014, the [STEIIAR](https://stellar-group.org)³⁴³⁸ 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](https://www.boost.org/)³⁴³⁹ 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 [STEIIAR](https://stellar-group.org)³⁴⁴⁰ and ParalleX, see *People* and *Why HPX?*.

2.11.2 People

The [STEIIAR](https://stellar-group.org)³⁴⁴¹ Group (pronounced as stellar) stands for “Systems Technology, Emergent Parallelism, and Algorithm Research”. We are an international group of faculty, researchers, and students working at various institutions around the world. The goal of the [STEIIAR](https://stellar-group.org)³⁴⁴² Group is to promote the development of scalable parallel applications by providing a community for ideas, a framework for collaboration, and a platform for communicating these concepts to the broader community.

Our work is focused on building technologies for scalable parallel applications. *HPX*, our general purpose C++ runtime system for parallel and distributed applications, is no exception. We use *HPX* for a broad range of scientific applications, helping scientists and developers to write code which scales better and shows better performance compared to more conventional programming models such as MPI.

HPX is based on *ParalleX* which is a new (and still experimental) parallel execution model aiming to overcome the limitations imposed by the current hardware and the techniques we use to write applications today. Our group focuses on two types of applications - those requiring excellent strong scaling, allowing for a dramatic reduction of execution time for fixed workloads and those needing highest level of sustained performance through massive parallelism. These applications are presently unable (through conventional practices) to effectively exploit a relatively small number of cores in a multi-core system. By extension, these application will not be able to exploit high-end exascale computing systems which are likely to employ hundreds of millions of such cores by the end of this decade.

Critical bottlenecks to the effective use of new generation high performance computing (HPC) systems include:

- *Starvation*: due to lack of usable application parallelism and means of managing it,
- *Overhead*: reduction to permit strong scalability, improve efficiency, and enable dynamic resource management,
- *Latency*: from remote access across system or to local memories,

³⁴³⁶ <https://www.lsu.edu>

³⁴³⁷ <https://stellar-group.org>

³⁴³⁸ <https://stellar-group.org>

³⁴³⁹ <https://www.boost.org/>

³⁴⁴⁰ <https://stellar-group.org>

³⁴⁴¹ <https://stellar-group.org>

³⁴⁴² <https://stellar-group.org>

- *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) ³⁴⁴³ , Louisiana State University (LSU) ³⁴⁴⁴	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture ³⁴⁴⁵ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ³⁴⁴⁶	thom.heller@gmail.com
Agustin Berge	Center for Computation and Technology (CCT) ³⁴⁴⁷ , Louisiana State University (LSU) ³⁴⁴⁸	kaballo86@hotmail.com
Mikael Simberg	Swiss National Supercomputing Centre ³⁴⁴⁹	simbergm@cscs.ch
John Biddiscombe	Swiss National Supercomputing Centre ³⁴⁵⁰	biddisco@cscs.ch
Anton Bikiineev	Center for Computation and Technology (CCT) ³⁴⁵¹ , Louisiana State University (LSU) ³⁴⁵²	ant.bikineev@gmail.com
Martin Stumpf	Department of Computer Science 3 - Computer Architecture ³⁴⁵³ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ³⁴⁵⁴	martin.h.stumpf@gmail.com
Bryce Adelstein Lelbach	NVIDIA ³⁴⁵⁵	brycelbach@gmail.com
Shuangyang Yang	Center for Computation and Technology (CCT) ³⁴⁵⁶ , Louisiana State University (LSU) ³⁴⁵⁷	syang16@cct.lsu.edu
Jeroen Habraken	Technische Universiteit Eindhoven ³⁴⁵⁸	jhabraken@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT) ³⁴⁵⁹ , Louisiana State University (LSU) ³⁴⁶⁰	sbrandt@cct.lsu.edu
Antoine Tran Tan	Center for Computation and Technology (CCT) ³⁴⁶¹ , Louisiana State University (LSU) ³⁴⁶²	antoine.trantan@lri.fr
Adrian Serio	Center for Computation and Technology (CCT) ³⁴⁶³ , Louisiana State University (LSU) ³⁴⁶⁴	aserio@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST) ³⁴⁶⁵ , Indiana University (IU) ³⁴⁶⁶	mbrodowi@indiana.edu

Contributors to this document

Table 2.40: Documentation authors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT) ³⁴⁶⁷ , Louisiana State University (LSU) ³⁴⁶⁸	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture ³⁴⁶⁹ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ³⁴⁷⁰	thom.heller@gmail.com
Bryce Adelstein Lelbach	NVIDIA ³⁴⁷¹	brycelelbach@gmail.com
Vinay C Amatya	Center for Computation and Technology (CCT) ³⁴⁷² , Louisiana State University (LSU) ³⁴⁷³	vamatya@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT) ³⁴⁷⁴ , Louisiana State University (LSU) ³⁴⁷⁵	sbrandt@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST) ³⁴⁷⁶ , Indiana University (IU) ³⁴⁷⁷	mbrodowi@indiana.edu
Adrian Serio	Center for Computation and Technology (CCT) ³⁴⁷⁸ , Louisiana State University (LSU) ³⁴⁷⁹	aserio@cct.lsu.edu

³⁴⁴³ <https://www.cct.lsu.edu>

³⁴⁴⁴ <https://www.lsu.edu>

³⁴⁴⁵ <https://www3.cs.fau.de>

³⁴⁴⁶ <https://www.fau.de>

³⁴⁴⁷ <https://www.cct.lsu.edu>

³⁴⁴⁸ <https://www.lsu.edu>

³⁴⁴⁹ <https://www.cscs.ch>

³⁴⁵⁰ <https://www.cscs.ch>

³⁴⁵¹ <https://www.cct.lsu.edu>

³⁴⁵² <https://www.lsu.edu>

³⁴⁵³ <https://www3.cs.fau.de>

³⁴⁵⁴ <https://www.fau.de>

³⁴⁵⁵ <https://nvidia.com/>

³⁴⁵⁶ <https://www.cct.lsu.edu>

³⁴⁵⁷ <https://www.lsu.edu>

³⁴⁵⁸ <https://www.tui.nl>

³⁴⁵⁹ <https://www.cct.lsu.edu>

³⁴⁶⁰ <https://www.lsu.edu>

³⁴⁶¹ <https://www.cct.lsu.edu>

³⁴⁶² <https://www.lsu.edu>

³⁴⁶³ <https://www.cct.lsu.edu>

³⁴⁶⁴ <https://www.lsu.edu>

³⁴⁶⁵ <https://pti.iu.edu>

³⁴⁶⁶ <https://www.iu.edu>

³⁴⁶⁷ <https://www.cct.lsu.edu>

³⁴⁶⁸ <https://www.lsu.edu>

³⁴⁶⁹ <https://www3.cs.fau.de>

³⁴⁷⁰ <https://www.fau.de>

³⁴⁷¹ <https://nvidia.com/>

³⁴⁷² <https://www.cct.lsu.edu>

³⁴⁷³ <https://www.lsu.edu>

³⁴⁷⁴ <https://www.cct.lsu.edu>

³⁴⁷⁵ <https://www.lsu.edu>

³⁴⁷⁶ <https://pti.iu.edu>

³⁴⁷⁷ <https://www.iu.edu>

³⁴⁷⁸ <https://www.cct.lsu.edu>

³⁴⁷⁹ <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](https://www.cscs.ch)³⁴⁸⁰), 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 \(LSU\)](https://www.lsu.edu)³⁴⁸¹), 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 \(LSU\)](https://www.lsu.edu)³⁴⁸²), 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 \(LSU\)](https://www.lsu.edu)³⁴⁸³), 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.

³⁴⁸⁰ <https://www.cscs.ch>

³⁴⁸¹ <https://www.lsu.edu>

³⁴⁸² <https://www.lsu.edu>

³⁴⁸³ <https://www.lsu.edu>

- Bibek Wagle ([Louisiana State University \(LSU\)](https://www.lsu.edu)³⁴⁸⁴), 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](https://www.libgeodecomp.org/)³⁴⁸⁵) with *HPX*. He reported various problems and submitted several patches to fix issues allowing for a better integration with [LibGeoDecomp](https://www.libgeodecomp.org/)³⁴⁸⁶.
- 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 \(LSU\)](https://www.lsu.edu)³⁴⁸⁷), 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](https://www.conan.io/)³⁴⁸⁸ C/C++ package manager.
- Vinay C Amatya ([Louisiana State University \(LSU\)](https://www.lsu.edu)³⁴⁸⁹), who contributed to the documentation and provided some of the *HPX* examples.
- Kevin Huck and Nick Chaimov ([University of Oregon](https://uoregon.edu/)³⁴⁹⁰), who contributed the integration of APEX (Automatic 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](https://github.com/STELLAR-GROUP/hpxcl/)³⁴⁹¹).
- 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 \(CCT\)](https://www.cct.lsu.edu)³⁴⁹²) who contributed to *HPX* the implementation of several parallel algorithms (as proposed by [N4313](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html)³⁴⁹³).
- Anuj Sharma and Christopher Bross ([Department of Computer Science 3 - Computer Architecture](https://www3.cs.fau.de)³⁴⁹⁴), who worked on *HPX* in the context of the [Google Summer of Code](https://developers.google.com/open-source/soc/)³⁴⁹⁵ program 2014.
- Martin Stumpf ([Department of Computer Science 3 - Computer Architecture](https://www3.cs.fau.de)³⁴⁹⁶), who rebuilt our contiguous testing infrastructure (see the [HPX Buildbot Website](https://rostam.cct.lsu.edu/)³⁴⁹⁷). Martin is also working on [HPXCL](https://github.com/STELLAR-GROUP/hpxcl/)³⁴⁹⁸ (mainly all

³⁴⁸⁴ <https://www.lsu.edu>

³⁴⁸⁵ <https://www.libgeodecomp.org/>

³⁴⁸⁶ <https://www.libgeodecomp.org/>

³⁴⁸⁷ <https://www.lsu.edu>

³⁴⁸⁸ <https://www.conan.io/>

³⁴⁸⁹ <https://www.lsu.edu>

³⁴⁹⁰ <https://uoregon.edu/>

³⁴⁹¹ <https://github.com/STELLAR-GROUP/hpxcl/>

³⁴⁹² <https://www.cct.lsu.edu>

³⁴⁹³ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

³⁴⁹⁴ <https://www3.cs.fau.de>

³⁴⁹⁵ <https://developers.google.com/open-source/soc/>

³⁴⁹⁶ <https://www3.cs.fau.de>

³⁴⁹⁷ <https://rostam.cct.lsu.edu/>

³⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpxcl/>

work related to [OpenCL](https://www.khronos.org/opencl/)³⁴⁹⁹) and implementing an *HPX* backend for [POCL](https://portablecl.org/)³⁵⁰⁰, a portable computing language solution based on [OpenCL](https://www.khronos.org/opencl/)³⁵⁰¹.

- Grant Mercer ([University of Nevada, Las Vegas](https://www.unlv.edu)³⁵⁰²), who helped creating many of the parallel algorithms (as proposed by [N4313](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html)³⁵⁰³).
- Damond Howard ([Louisiana State University \(LSU\)](https://www.lsu.edu)³⁵⁰⁴), who works on [HPXCL](https://www.nvidia.com/object/cuda_home_new.html)³⁵⁰⁵ (mainly all work related to [CUDA](https://www.nvidia.com/object/cuda_home_new.html)³⁵⁰⁶).
- 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](https://www.numscale.com/nt2/)³⁵⁰⁷. He also contributed an implementation of an API similar to Fortran co-arrays on top of *HPX*.
- John Biddiscombe ([Swiss National Supercomputing Centre](https://www.cscs.ch)³⁵⁰⁸), 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](https://www.nmsu.edu)³⁵⁰⁹), 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](https://www.helvihartmann.com), Nicholas Pezolano, and John West who added and improved examples in *HPX*.

³⁴⁹⁹ <https://www.khronos.org/opencl/>

³⁵⁰⁰ <https://portablecl.org/>

³⁵⁰¹ <https://www.khronos.org/opencl/>

³⁵⁰² <https://www.unlv.edu>

³⁵⁰³ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

³⁵⁰⁴ <https://www.lsu.edu>

³⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpxcl/>

³⁵⁰⁶ https://www.nvidia.com/object/cuda_home_new.html

³⁵⁰⁷ <https://www.numscale.com/nt2/>

³⁵⁰⁸ <https://www.cscs.ch>

³⁵⁰⁹ <https://www.nmsu.edu>

- Joseph Kleinhenz, Markus Elfring, Kirill Kropivnyansky, 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 \(CCT\)](https://www.cct.lsu.edu)³⁵¹⁰, [Department of Computer Science 3 - Computer Architecture](https://www3.cs.fau.de)³⁵¹¹, and [Swiss National Supercomputing Centre](https://www.cscs.ch)³⁵¹² 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 Forschungsstiftung) through the grant AZ-987-11.
- The European Commission's Horizon 2020 programme through the grant H2020-EU.1.2.2. 671603 (AllScale).

³⁵¹⁰ <https://www.cct.lsu.edu>

³⁵¹¹ <https://www3.cs.fau.de>

³⁵¹² <https://www.cscs.ch>

CHAPTER 3

Index

- `genindex`

Symbols

- hpx:affinity arg
 - command line option, 119
- hpx:agas arg
 - command line option, 118
- hpx:app-config arg
 - command line option, 119
- hpx:attach-debugger arg
 - command line option, 120
- hpx:bind arg
 - command line option, 119
- hpx:config arg
 - command line option, 119
- hpx:connect
 - command line option, 118
- hpx:console
 - command line option, 118
- hpx:cores arg
 - command line option, 119
- hpx:debug-agas-log [arg]
 - command line option, 120
- hpx:debug-app-log [arg]
 - command line option, 120
- hpx:debug-clp
 - command line option, 120
- hpx:debug-hpx-log [arg]
 - command line option, 120
- hpx:debug-parcel-log [arg]
 - command line option, 120
- hpx:debug-timing-log [arg]
 - command line option, 120
- hpx:dump-config
 - command line option, 120
- hpx:dump-config-initial
 - command line option, 120
- hpx:endnodes
 - command line option, 118
- hpx:exit
 - command line option, 120
- hpx:expect-connecting-localities
 - command line option, 119
- hpx:help
 - command line option, 118
- hpx:high-priority-threads arg
 - command line option, 119
- hpx:hpx arg
 - command line option, 118
- hpx:ifprefix arg
 - command line option, 118
- hpx:ifsuffix arg
 - command line option, 118
- hpx:iftransform arg
 - command line option, 118
- hpx:ignore-batch-env
 - command line option, 119
- hpx:info
 - command line option, 118
- hpx:ini arg
 - command line option, 120
- hpx:list-component-types
 - command line option, 120
- hpx:list-counter-infos
 - command line option, 121
- hpx:list-counters
 - command line option, 121
- hpx:list-symbolic-names
 - command line option, 120
- hpx:localities arg
 - command line option, 118
- hpx:no-csv-header
 - command line option, 121
- hpx:node arg
 - command line option, 119
- hpx:nodefile arg
 - command line option, 118
- hpx:nodes arg
 - command line option, 118
- hpx:numa-sensitive
 - command line option, 119

- hpx:options-file arg
 - command line option, [118](#)
- hpx:print-bind
 - command line option, [119](#)
- hpx:print-counter
 - command line option, [120](#)
- hpx:print-counter-at arg
 - command line option, [121](#)
- hpx:print-counter-destination
 - command line option, [121](#)
- hpx:print-counter-format
 - command line option, [121](#)
- hpx:print-counter-interval
 - command line option, [120](#)
- hpx:print-counter-reset
 - command line option, [120](#)
- hpx:print-counters-locally
 - command line option, [121](#)
- hpx:pu-offset
 - command line option, [119](#)
- hpx:pu-step
 - command line option, [119](#)
- hpx:queuing arg
 - command line option, [119](#)
- hpx:reset-counters
 - command line option, [121](#)
- hpx:run-agas-server
 - command line option, [118](#)
- hpx:run-agas-server-only
 - command line option, [118](#)
- hpx:run-hpx-main
 - command line option, [118](#)
- hpx:threads arg
 - command line option, [119](#)
- hpx:version
 - command line option, [118](#)
- hpx:worker
 - command line option, [118](#)

A

- Action, [18](#)
- Active Global Address Space, [17](#)
- AGAS, [17](#)
- AMPLIFIER_ROOT:PATH
 - command line option, [80](#)
- applier (C++ type), [250](#)

B

- BOOST_ROOT:PATH
 - command line option, [79](#)
- BREATHE_APIDOC_ROOT:PATH
 - command line option, [45](#)

C

- command line option
 - hpx:affinity arg, [119](#)
 - hpx:agas arg, [118](#)
 - hpx:app-config arg, [119](#)
 - hpx:attach-debugger arg, [120](#)
 - hpx:bind arg, [119](#)
 - hpx:config arg, [119](#)
 - hpx:connect, [118](#)
 - hpx:console, [118](#)
 - hpx:cores arg, [119](#)
 - hpx:debug-agas-log [arg], [120](#)
 - hpx:debug-app-log [arg], [120](#)
 - hpx:debug-clp, [120](#)
 - hpx:debug-hpx-log [arg], [120](#)
 - hpx:debug-parcel-log [arg], [120](#)
 - hpx:debug-timing-log [arg], [120](#)
 - hpx:dump-config, [120](#)
 - hpx:dump-config-initial, [120](#)
 - hpx:endnodes, [118](#)
 - hpx:exit, [120](#)
 - hpx:expect-connecting-localities, [119](#)
 - hpx:help, [118](#)
 - hpx:high-priority-threads arg, [119](#)
 - hpx:hpx arg, [118](#)
 - hpx:ifprefix arg, [118](#)
 - hpx:ifsuffix arg, [118](#)
 - hpx:iftransform arg, [118](#)
 - hpx:ignore-batch-env, [119](#)
 - hpx:info, [118](#)
 - hpx:ini arg, [120](#)
 - hpx:list-component-types, [120](#)
 - hpx:list-counter-infos, [121](#)
 - hpx:list-counters, [121](#)
 - hpx:list-symbolic-names, [120](#)
 - hpx:localities arg, [118](#)
 - hpx:no-csv-header, [121](#)
 - hpx:node arg, [119](#)
 - hpx:nodefile arg, [118](#)
 - hpx:nodes arg, [118](#)
 - hpx:numa-sensitive, [119](#)
 - hpx:options-file arg, [118](#)
 - hpx:print-bind, [119](#)
 - hpx:print-counter, [120](#)
 - hpx:print-counter-at arg, [121](#)
 - hpx:print-counter-destination, [121](#)
 - hpx:print-counter-format, [121](#)
 - hpx:print-counter-interval, [120](#)
 - hpx:print-counter-reset, [120](#)
 - hpx:print-counters-locally, [121](#)
 - hpx:pu-offset, [119](#)
 - hpx:pu-step, [119](#)
 - hpx:queuing arg, [119](#)

-hpx:reset-counters, 121
 -hpx:run-agas-server, 118
 -hpx:run-agas-server-only, 118
 -hpx:run-hpx-main, 118
 -hpx:threads arg, 119
 -hpx:version, 118
 -hpx:worker, 118
 AMPLIFIER_ROOT:PATH, 80
 BOOST_ROOT:PATH, 79
 BREATHE_APIDOC_ROOT:PATH, 45
 DOXYGEN_ROOT:PATH, 45
 HDF5_ROOT:PATH, 80
 HPX_SCHEDULER_MAX_TERMINATED_THREADS:STRING, 76
 HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL, 77
 HPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT:BOOL, 72
 HPX_WITH_APEX:BOOL, 78
 HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL, 79
 HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL, 72
 HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH, 72
 HPX_WITH_BUILD_BINARY_PACKAGE:BOOL, 72
 HPX_WITH_COMPILE_ONLY_TESTS:BOOL, 74
 HPX_WITH_COMPILER_WARNINGS:BOOL, 72
 HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL, 72
 HPX_WITH_COMPRESSION_BZIP2:BOOL, 72
 HPX_WITH_COMPRESSION_SNAPPY:BOOL, 72
 HPX_WITH_COMPRESSION_ZLIB:BOOL, 72
 HPX_WITH_CUDA:BOOL, 72
 HPX_WITH_CUDA_CLANG:BOOL, 72
 HPX_WITH_CXX14_RETURN_TYPE_DEDUCTION:BOOL, 72
 HPX_WITH_DATAPAR_BOOST_SIMD:BOOL, 72
 HPX_WITH_DATAPAR_VC:BOOL, 72
 HPX_WITH_DEFAULT_TARGETS:BOOL, 74
 HPX_WITH_DEPRECATION_WARNINGS:BOOL, 72
 HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL, 72
 HPX_WITH_DOCUMENTATION:BOOL, 74
 HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS:STRING, 74
 HPX_WITH_DYNAMIC_HPX_MAIN:BOOL, 72
 HPX_WITH_EXAMPLES:BOOL, 74
 HPX_WITH_EXAMPLES_HDF5:BOOL, 74
 HPX_WITH_EXAMPLES_OPENMP:BOOL, 74
 HPX_WITH_EXAMPLES_QT4:BOOL, 74
 HPX_WITH_EXAMPLES_QTHREADS:BOOL, 75
 HPX_WITH_EXAMPLES_TBB:BOOL, 75
 HPX_WITH_EXECUTABLE_PREFIX:STRING, 75
 HPX_WITH_FAIL_COMPILE_TESTS:BOOL, 75
 HPX_WITH_FAULT_TOLERANCE:BOOL, 72
 HPX_WITH_FORTRAN:BOOL, 72
 HPX_WITH_FULL_RPATH:BOOL, 72
 HPX_WITH_GCC_VERSION_CHECK:BOOL, 73
 HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL, 73
 HPX_WITH_GOOGLE_PERFTOOLS:BOOL, 78
 HPX_WITH_HCC:BOOL, 73
 HPX_WITH_HIDDEN_VISIBILITY:BOOL, 73
 HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY:BOOL, 73
 HPX_WITH_IO_COUNTERS:BOOL, 75
 HPX_WITH_IO_POOL:BOOL, 76
 HPX_WITH_ITTNOTIFY:BOOL, 78
 HPX_WITH_LOGGING:BOOL, 73
 HPX_WITH_MALLOC:STRING, 73
 HPX_WITH_MAX_CPU_COUNT:STRING, 76
 HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING, 76
 HPX_WITH_MORE_THAN_64_THREADS:BOOL, 76
 HPX_WITH_NATIVE_TLS:BOOL, 73
 HPX_WITH_NETWORKING:BOOL, 77
 HPX_WITH_NICE_THREADLEVEL:BOOL, 73
 HPX_WITH_PAPI:BOOL, 78
 HPX_WITH_PARCEL_COALESCING:BOOL, 73
 HPX_WITH_PARCEL_PROFILING:BOOL, 78
 HPX_WITH_PARCELPART_ACTION_COUNTERS:BOOL, 77
 HPX_WITH_PARCELPART_LIBFABRIC:BOOL, 78
 HPX_WITH_PARCELPART_MPI:BOOL, 78
 HPX_WITH_PARCELPART_MPI_ENV:STRING, 78
 HPX_WITH_PARCELPART_MPI_MULTITHREADED:BOOL, 78
 HPX_WITH_PARCELPART_TCP:BOOL, 78
 HPX_WITH_PARCELPART_VERBS:BOOL, 78
 HPX_WITH_PSEUDO_DEPENDENCIES:BOOL, 75
 HPX_WITH_QUEUE_COMPATIBILITY:BOOL, 73
 HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL, 73
 HPX_WITH_SANITIZERS:BOOL, 79
 HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL, 76
 HPX_WITH_SCOPED_UNLOCK_COMPATIBILITY:BOOL, 73

HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL, 76
 HPX_WITH_SPINLOCK_POOL_NUM:STRING, 76
 HPX_WITH_STACKOVERFLOW_DETECTION:BOOL, 73
 HPX_WITH_STACKTRACES:BOOL, 76
 HPX_WITH_STATIC_LINKING:BOOL, 73
 HPX_WITH_SWAP_CONTEXT_EMULATION:BOOL, 76
 HPX_WITH_SYCL:BOOL, 73
 HPX_WITH_TESTS:BOOL, 75
 HPX_WITH_TESTS_BENCHMARKS:BOOL, 75
 HPX_WITH_TESTS_DEBUG_LOG:BOOL, 79
 HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING, 79
 HPX_WITH_TESTS_EXAMPLES:BOOL, 75
 HPX_WITH_TESTS_EXTERNAL_BUILD:BOOL, 75
 HPX_WITH_TESTS_HEADERS:BOOL, 75
 HPX_WITH_TESTS_REGRESSIONS:BOOL, 75
 HPX_WITH_TESTS_UNIT:BOOL, 75
 HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING, 76
 HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL, 76
 HPX_WITH_THREAD_COMPATIBILITY:BOOL, 73
 HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL, 76
 HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL, 76
 HPX_WITH_THREAD_DEBUG_INFO:BOOL, 79
 HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL, 79
 HPX_WITH_THREAD_GUARD_PAGE:BOOL, 79
 HPX_WITH_THREAD_IDLE_RATES:BOOL, 77
 HPX_WITH_THREAD_LOCAL_STORAGE:BOOL, 77
 HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL, 77
 HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL, 77
 HPX_WITH_THREAD_SCHEDULERS:STRING, 77
 HPX_WITH_THREAD_STACK_MMAP:BOOL, 77
 HPX_WITH_THREAD_STEALING_COUNTS:BOOL, 77
 HPX_WITH_THREAD_TARGET_ADDRESS:BOOL, 77
 HPX_WITH_TIMER_POOL:BOOL, 77
 HPX_WITH_TOOLS:BOOL, 75
 HPX_WITH_UNWRAPPED_COMPATIBILITY:BOOL, 73
 HPX_WITH_VALGRIND:BOOL, 79
 HPX_WITH_VERIFY_LOCKS:BOOL, 79
 HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL, 79
 HPX_WITH_VERIFY_LOCKS_GLOBALLY:BOOL, 79
 HPX_WITH_VIM_YCM:BOOL, 73
 HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING, 73
 HWLOC_ROOT:PATH, 79
 PAPI_ROOT:PATH, 80
 SPHINX_ROOT:PATH, 45
 Component, 18
D
 DOXYGEN_ROOT:PATH
 command line option, 45
H
 HDF5_ROOT:PATH
 command line option, 80
 hpx (C++ type), 250
 hpx::actions (C++ type), 322
 hpx::applier (C++ type), 322
 hpx::applier::get_applier (C++ function), 322
 hpx::applier::get_applier_ptr (C++ function), 322
 hpx::assertion_failure (C++ enumerator), 252
 hpx::bad_action_code (C++ enumerator), 251
 hpx::bad_component_type (C++ enumerator), 251
 hpx::bad_function_call (C++ enumerator), 253
 hpx::bad_parameter (C++ enumerator), 251
 hpx::bad_plugin_type (C++ enumerator), 253
 hpx::bad_request (C++ enumerator), 251
 hpx::bad_response_type (C++ enumerator), 252
 hpx::broken_promise (C++ enumerator), 252
 hpx::broken_task (C++ enumerator), 252
 hpx::commandline_option_error (C++ enumerator), 252
 hpx::components (C++ type), 322
 hpx::components::binpacked (C++ member), 326
 hpx::components::binpacking_distribution_policy (C++ class), 209
 hpx::components::binpacking_distribution_policy::binpacking_distribution_policy (C++ function), 209
 hpx::components::binpacking_distribution_policy::binpacking_distribution_policy (C++ function), 210
 hpx::components::binpacking_distribution_policy::binpacking_distribution_policy (C++ function), 210
 hpx::components::binpacking_distribution_policy::binpacking_distribution_policy (C++ function), 210

hpx::components::binpacking_distribution_policy(C++ function), 136
 (C++ function), 210
 hpx::components::binpacking_distribution_policy(C++ function), 216
 (C++ function), 209, 210
 hpx::components::colocated (C++ member), (C++ function), 216
 326
 hpx::components::colocating_distribution_policy(C++ function), 215
 (C++ class), 212
 hpx::components::colocating_distribution_policy(member), 326
 (C++ function), 214
 hpx::components::colocating_distribution_policy(apply_cb
 (C++ function), 214
 hpx::components::colocating_distribution_policy(C++ function), 323
 (C++ function), 213
 hpx::components::colocating_distribution_policy(C++ function), 323
 (C++ function), 213
 hpx::components::colocating_distribution_policy(async_result
 (C++ class), 207
 hpx::components::colocating_distribution_policy(id_async_incrementation(C++ function),
 (C++ type), 207 275
 hpx::components::colocating_distribution_policy(id_is_blocked(C++ function), 273, 274
 (C++ function), 213
 hpx::components::colocating_distribution_policy(enumerator), 251
 (C++ function), 213
 hpx::components::colocating_distribution_policy(create
 (C++ function), 213
 hpx::components::colocating_distribution_policy(dynamic_get_max_fairshare(C++ enumerator),
 (C++ function), 214 252
 hpx::components::colocating_distribution_policy(policy(C++ enum), 251
 (C++ function), 214
 hpx::components::colocating_distribution_policy(policy_code(C++ function), 219
 (C++ function), 213
 hpx::components::copy (C++ function), 324 217–219
 hpx::components::default_binpacking_count_hpx::error_code::exception_ (C++ member),
 (C++ member), 326 219
 hpx::components::default_distribution_policy_hpx::error_code::get_message (C++ func-
 (C++ class), 215 tion), 219
 hpx::components::default_distribution_policy_hpx::applycode::operator= (C++ function),
 (C++ function), 216 219
 hpx::components::default_distribution_policy_hpx::applyactive_counters (C++ func-
 (C++ function), 216 tion), 285
 hpx::components::default_distribution_policy_hpx::exception (C++ class), 220
 (C++ function), 216
 hpx::components::default_distribution_policy::async_cb
 (C++ function), 216
 hpx::components::default_distribution_policy::exception (C++ function),
 (C++ class), 207
 hpx::components::default_distribution_policy::async_result
 (C++ type), 208
 hpx::components::default_distribution_policy::create
 (C++ function), 215
 hpx::components::default_distribution_policy::exception_list (C++ class), 221
 (C++ function), 215
 hpx::components::default_distribution_policy::exception_list::begin (C++ function),
 (C++ function), 215 221
 hpx::components::default_distribution_policy::exception_list::end (C++ function), 221

`hpx::exception_list::exception_list_type` `hpx::get_hpx_rethrow_category` (C++ function), 275
(C++ type), 222
`hpx::exception_list::exceptions_` (C++ member), 222
`hpx::exception_list::iterator` (C++ type), 221
`hpx::exception_list::mtx_` (C++ member), 222
`hpx::exception_list::mutex_type` (C++ type), 222
`hpx::exception_list::size` (C++ function), 221
`hpx::filesystem_error` (C++ enumerator), 253
`hpx::finalize` (C++ function), 272, 273
`hpx::find_all_from_basename` (C++ function), 285
`hpx::find_all_localities` (C++ function), 289, 290
`hpx::find_from_basename` (C++ function), 286, 287
`hpx::find_here` (C++ function), 289
`hpx::find_locality` (C++ function), 291
`hpx::find_remote_localities` (C++ function), 290, 291
`hpx::find_root_locality` (C++ function), 289
`hpx::future_already_retrieved` (C++ enumerator), 252
`hpx::future_can_not_be_cancelled` (C++ enumerator), 252
`hpx::future_cancelled` (C++ enumerator), 252
`hpx::future_does_not_support_cancellation` (C++ enumerator), 252
`hpx::get_colocation_id` (C++ function), 292
`hpx::get_error` (C++ function), 277
`hpx::get_error_backtrace` (C++ function), 279
`hpx::get_error_config` (C++ function), 282
`hpx::get_error_env` (C++ function), 278
`hpx::get_error_file_name` (C++ function), 279
`hpx::get_error_function_name` (C++ function), 279
`hpx::get_error_host_name` (C++ function), 277
`hpx::get_error_line_number` (C++ function), 280
`hpx::get_error_locality_id` (C++ function), 276
`hpx::get_error_os_thread` (C++ function), 280
`hpx::get_error_process_id` (C++ function), 278
`hpx::get_error_state` (C++ function), 282
`hpx::get_error_thread_description` (C++ function), 281
`hpx::get_error_thread_id` (C++ function), 281
`hpx::get_error_what` (C++ function), 276
`hpx::get_hpx_category` (C++ function), 275
`hpx::get_initial_num_localities` (C++ function), 295
`hpx::get_locality` (C++ function), 283
`hpx::get_locality_id` (C++ function), 294
`hpx::get_locality_name` (C++ function), 295
`hpx::get_num_localities` (C++ function), 295, 296
`hpx::get_num_worker_threads` (C++ function), 283
`hpx::get_os_thread_count` (C++ function), 296
`hpx::get_ptr` (C++ function), 292–294
`hpx::get_runtime_instance_number` (C++ function), 283
`hpx::get_runtime_mode_from_name` (C++ function), 297
`hpx::get_runtime_mode_name` (C++ function), 297
`hpx::get_system_uptime` (C++ function), 284
`hpx::get_thread_name` (C++ function), 296
`hpx::get_worker_thread_num` (C++ function), 296, 297
`hpx::init` (C++ function), 254–262
`hpx::internal_server_error` (C++ enumerator), 251
`hpx::invalid_data` (C++ enumerator), 252
`hpx::invalid_status` (C++ enumerator), 251
`hpx::is_running` (C++ function), 283
`hpx::is_starting` (C++ function), 283
`hpx::is_stopped` (C++ function), 283
`hpx::is_stopped_or_shutting_down` (C++ function), 283
`hpx::kernel_error` (C++ enumerator), 252
`hpx::launch` (C++ class), 223
`hpx::launch::apply` (C++ member), 224
`hpx::launch::deferred` (C++ member), 224
`hpx::launch::fork` (C++ member), 224
`hpx::launch::launch` (C++ function), 223
`hpx::launch::select` (C++ member), 224
`hpx::launch::sync` (C++ member), 224
`hpx::lcos` (C++ type), 326
`hpx::lcos::barrier` (C++ class), 208
`hpx::lcos::barrier::barrier` (C++ function), 208, 209
`hpx::lcos::barrier::synchronize` (C++ function), 209
`hpx::lcos::barrier::wait` (C++ function), 209
`hpx::lcos::gather_here` (C++ function), 330, 331
`hpx::lcos::gather_there` (C++ function), 330, 331
`hpx::length_error` (C++ enumerator), 253
`hpx::lock_error` (C++ enumerator), 251

hpx::make_error_code (C++ function), 250	hpx::parallel::execution::parallel_policy::executor (C++ type), 225
hpx::make_success_code (C++ function), 275	hpx::parallel::execution::parallel_policy::executor (C++ type), 225
hpx::migration_needs_retry (C++ enumerator), 253	hpx::parallel::execution::parallel_policy::on (C++ function), 225
hpx::naming (C++ type), 332	hpx::parallel::execution::parallel_policy::operator (C++ function), 225
hpx::naming::unmanaged (C++ function), 333	hpx::parallel::execution::parallel_policy::parameter (C++ function), 226
hpx::network_error (C++ enumerator), 251	hpx::parallel::execution::parallel_policy::params (C++ member), 226
hpx::no_registered_console (C++ enumerator), 251	hpx::parallel::execution::parallel_policy::rebind (C++ class), 235
hpx::no_state (C++ enumerator), 252	hpx::parallel::execution::parallel_policy::rebind (C++ type), 235
hpx::no_success (C++ enumerator), 251	hpx::parallel::execution::parallel_policy::serializable (C++ function), 226
hpx::not_implemented (C++ enumerator), 251	hpx::parallel::execution::parallel_policy::with (C++ function), 225
hpx::null_thread_id (C++ enumerator), 252	hpx::parallel::execution::parallel_policy_executor (C++ class), 226
hpx::out_of_memory (C++ enumerator), 251	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::out_of_range (C++ enumerator), 253	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel (C++ type), 333	hpx::parallel::execution::parallel_policy_executor (C++ function), 226
hpx::parallel::execution (C++ type), 333	hpx::parallel::execution::parallel_policy_executor (C++ function), 226
hpx::parallel::execution::auto_chunk_size (C++ class), 208	hpx::parallel::execution::parallel_policy_executor (C++ function), 226
hpx::parallel::execution::auto_chunk_size (C++ function), 208	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::dynamic_chunk_size (C++ class), 216	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::dynamic_chunk_size (C++ function), 216	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::guided_chunk_size (C++ class), 222	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::guided_chunk_size (C++ function), 222	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::io_pool_executor (C++ type), 333	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::is_async_executor (C++ class), 222	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::is_execution_parallel (C++ class), 223	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::is_parallel_executor (C++ class), 223	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::is_sequenced_executor (C++ class), 223	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::local_priority_executor (C++ type), 333	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::main_pool_executor (C++ type), 333	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::parallel_executor (C++ class), 224	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::parallel_executor (C++ type), 333	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::parallel_policy (C++ class), 224	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::parallel_policy (C++ member), 226	hpx::parallel::execution::parallel_policy_executor (C++ type), 226
hpx::parallel::execution::parallel_policy::executor (C++ type), 225	hpx::parallel::execution::parallel_policy_executor (C++ type), 226

[illegible]

(C++ class), 238
 hpx::parallel::execution::sequenced_policy_shim(C++ function), 242
 (C++ type), 238
 hpx::parallel::execution::sequenced_policy_shim(C++ type), 241
 (C++ function), 239
 hpx::parallel::execution::sequenced_policy_shim(C++ type), 241 parameters_type
 (C++ type), 238
 hpx::parallel::execution::sequenced_policy_shim(C++ function), 241
 (C++ type), 238
 hpx::parallel::execution::sequenced_policy_shim(C++ function), 241
 (C++ function), 238
 hpx::parallel::execution::sequenced_policy_shim(C++ function), 242
 (C++ function), 238
 hpx::parallel::execution::sequenced_policy_shim(C++ class), 238
 (C++ function), 239
 hpx::parallel::execution::sequenced_policy_shim(C++ function), 242
 (C++ class), 236
 hpx::parallel::execution::sequenced_policy_shim(Parameters>::rebind::type (C++
 (C++ function), 238 type), 235
 hpx::parallel::execution::sequenced_policy_shim(Parameters>::rebind::type (C++
 Parameters>::rebind::type (C++ (C++ type), 333
 type), 236 hpx::parallel::execution::static_chunk_size
 hpx::parallel::execution::sequenced_task_policy(C++ class), 242
 (C++ class), 239 hpx::parallel::execution::static_chunk_size::static
 hpx::parallel::execution::sequenced_task_policy(C++ function), 242
 (C++ member), 241 hpx::parallel::execution::timer_pool_executor
 hpx::parallel::execution::sequenced_task_policy(C++ type), 333 category
 (C++ type), 239 hpx::parallel::execution::unsequenced_execution_tag
 hpx::parallel::execution::sequenced_task_policy(C++ class), 249
 (C++ function), 240 hpx::parallel::execution::[anonymous]
 hpx::parallel::execution::sequenced_task_policy(C++ type), 334 parameters_type
 (C++ type), 239 hpx::parallel::v1 (C++ type), 334
 hpx::parallel::execution::sequenced_task_policy(v1::adjacent_difference
 (C++ type), 239 (C++ function), 334, 335
 hpx::parallel::execution::sequenced_task_policy(v1::adjacent_find (C++
 (C++ function), 240 function), 336
 hpx::parallel::execution::sequenced_task_policy(v1::all_of (C++ function),
 (C++ function), 239 339, 431
 hpx::parallel::execution::sequenced_task_policy(v1::any_of (C++ function),
 (C++ function), 240 338, 430
 hpx::parallel::execution::sequenced_task_policy(v1::copy (C++ function), 340,
 (C++ member), 241 432
 hpx::parallel::execution::sequenced_task_policy(v1::copy_if (C++ function),
 (C++ class), 235 341, 432
 hpx::parallel::execution::sequenced_task_policy(v1::destroy_n (C++ function),
 (C++ type), 236 340
 hpx::parallel::execution::sequenced_task_policy(v1::count (C++ function), 342,
 (C++ function), 240 433
 hpx::parallel::execution::sequenced_task_policy(v1::count_if (C++ function),
 (C++ function), 240 343, 434
 hpx::parallel::execution::sequenced_task_policy(v1::destroy (C++ function),
 (C++ class), 241 344
 hpx::parallel::execution::sequenced_task_policy(v1::execute_at_least_once (C++ func-
 (C++ type), 241 tion), 345

`hpx::parallel::v1::equal (C++ function), 345, 346`
`hpx::parallel::v1::exclusive_scan (C++ function), 348, 349`
`hpx::parallel::v1::fill (C++ function), 350, 435`
`hpx::parallel::v1::fill_n (C++ function), 351, 436`
`hpx::parallel::v1::find (C++ function), 351`
`hpx::parallel::v1::find_end (C++ function), 354, 437`
`hpx::parallel::v1::find_first_of (C++ function), 355, 438`
`hpx::parallel::v1::find_if (C++ function), 352`
`hpx::parallel::v1::find_if_not (C++ function), 353`
`hpx::parallel::v1::for_each (C++ function), 358, 440`
`hpx::parallel::v1::for_each_n (C++ function), 357`
`hpx::parallel::v1::generate (C++ function), 359, 441`
`hpx::parallel::v1::generate_n (C++ function), 360`
`hpx::parallel::v1::includes (C++ function), 362`
`hpx::parallel::v1::inclusive_scan (C++ function), 363, 365, 366`
`hpx::parallel::v1::inplace_merge (C++ function), 372, 445`
`hpx::parallel::v1::is_heap (C++ function), 360, 441`
`hpx::parallel::v1::is_heap_until (C++ function), 361, 442`
`hpx::parallel::v1::is_partitioned (C++ function), 367`
`hpx::parallel::v1::is_sorted (C++ function), 367`
`hpx::parallel::v1::is_sorted_until (C++ function), 368`
`hpx::parallel::v1::lexicographical_compare (C++ function), 369`
`hpx::parallel::v1::max_element (C++ function), 374, 447`
`hpx::parallel::v1::merge (C++ function), 371, 443`
`hpx::parallel::v1::min_element (C++ function), 373, 446`
`hpx::parallel::v1::minmax_element (C++ function), 375, 448`
`hpx::parallel::v1::mismatch (C++ function), 376, 377`
`hpx::parallel::v1::move (C++ function), 378`
`hpx::parallel::v1::none_of (C++ function), 337, 429`
`hpx::parallel::v1::partition (C++ function), 380, 449`
`hpx::parallel::v1::partition_copy (C++ function), 381, 450`
`hpx::parallel::v1::reduce (C++ function), 382–384`
`hpx::parallel::v1::reduce_by_key (C++ function), 385`
`hpx::parallel::v1::remove (C++ function), 388, 451`
`hpx::parallel::v1::remove_copy (C++ function), 389, 453`
`hpx::parallel::v1::remove_copy_if (C++ function), 390, 454`
`hpx::parallel::v1::remove_if (C++ function), 387, 452`
`hpx::parallel::v1::replace (C++ function), 391, 455`
`hpx::parallel::v1::replace_copy (C++ function), 393, 457`
`hpx::parallel::v1::replace_copy_if (C++ function), 394, 458`
`hpx::parallel::v1::replace_if (C++ function), 392, 456`
`hpx::parallel::v1::reverse (C++ function), 395, 459`
`hpx::parallel::v1::reverse_copy (C++ function), 395, 460`
`hpx::parallel::v1::rotate (C++ function), 396, 461`
`hpx::parallel::v1::rotate_copy (C++ function), 397, 461`
`hpx::parallel::v1::search (C++ function), 397, 462`
`hpx::parallel::v1::search_n (C++ function), 399, 464`
`hpx::parallel::v1::set_difference (C++ function), 400`
`hpx::parallel::v1::set_intersection (C++ function), 401`
`hpx::parallel::v1::set_symmetric_difference (C++ function), 403`
`hpx::parallel::v1::set_union (C++ function), 405`
`hpx::parallel::v1::sort (C++ function), 406, 465`
`hpx::parallel::v1::sort_by_key (C++ function), 407`
`hpx::parallel::v1::stable_partition (C++ function), 379`
`hpx::parallel::v1::swap_ranges (C++ function), 408`

hpx::parallel::v1::transform (C++ function), 409–411, 467, 469
 hpx::parallel::v1::transform_exclusive_scan (C++ function), 412
 hpx::parallel::v1::transform_inclusive_scan (C++ function), 414, 415
 hpx::parallel::v1::transform_reduce (C++ function), 417–419
 hpx::parallel::v1::uninitialized_copy (C++ function), 420
 hpx::parallel::v1::uninitialized_copy_n (C++ function), 421
 hpx::parallel::v1::uninitialized_default_construct (C++ function), 422
 hpx::parallel::v1::uninitialized_default_construct_n (C++ function), 422
 hpx::parallel::v1::uninitialized_fill (C++ function), 423
 hpx::parallel::v1::uninitialized_fill_n (C++ function), 424
 hpx::parallel::v1::uninitialized_move (C++ function), 425
 hpx::parallel::v1::uninitialized_move_n (C++ function), 425
 hpx::parallel::v1::uninitialized_value_construct (C++ function), 426
 hpx::parallel::v1::uninitialized_value_construct_n (C++ function), 427
 hpx::parallel::v1::unique (C++ function), 427, 470
 hpx::parallel::v1::unique_copy (C++ function), 428, 471
 hpx::parallel::v2 (C++ type), 472
 hpx::parallel::v2::define_task_block (C++ function), 472, 473
 hpx::parallel::v2::define_task_block_reschedule (C++ function), 473, 474
 hpx::parallel::v2::for_loop (C++ function), 474, 475
 hpx::parallel::v2::induction (C++ function), 483
 hpx::parallel::v2::reduction (C++ function), 483
 hpx::parallel::v2::task_block (C++ class), 242
 hpx::parallel::v2::task_block::errors_ (C++ member), 245
 hpx::parallel::v2::task_block::execution_policy_ (C++ type), 243
 hpx::parallel::v2::task_block::get_execution_policy (C++ function), 243
 hpx::parallel::v2::task_block::id_ (C++ member), 245
 hpx::parallel::v2::task_block::mtx_ (C++ member), 245
 hpx::parallel::v2::task_block::policy_ (C++ member), 245
 hpx::parallel::v2::task_block::policy_ (C++ member), 245
 hpx::parallel::v2::task_block::run (C++ function), 243, 244
 hpx::parallel::v2::task_block::tasks_ (C++ member), 245
 hpx::parallel::v2::task_block::wait (C++ function), 244
 hpx::parallel::v2::task_canceled_exception (C++ class), 245
 hpx::parallel::v2::task_canceled_exception::task_cancelled (C++ function), 245
 hpx::parcel_write_handler_type (C++ type), 250
 hpx::performance_counters (C++ type), 484
 hpx::performance_counters::install_counter_type (C++ function), 484–486
 hpx::plain (C++ enumerator), 253
 hpx::promise_already_satisfied (C++ enumerator), 252
 hpx::register_on_exit (C++ function), 283
 hpx::register_pre_shutdown_function (C++ function), 297
 hpx::register_pre_startup_function (C++ function), 298
 hpx::register_shutdown_function (C++ function), 298
 hpx::register_startup_function (C++ function), 298
 hpx::register_thread (C++ function), 282
 hpx::register_with_basename (C++ function), 287, 288
 hpx::thread_active_counters (C++ function), 284
 hpx::repeated_request (C++ enumerator), 251
 hpx::report_error (C++ function), 297
 hpx::reset_active_counters (C++ function), 284
 hpx::resource (C++ type), 487
 hpx::resource::abp_priority_fifo (C++ enumerator), 488
 hpx::resource::abp_priority_lifo (C++ enumerator), 488
 hpx::resource::core (C++ class), 214
 hpx::resource::core::core (C++ function), 214
 hpx::resource::core::cores_sharing_numa_domain (C++ function), 214
 hpx::resource::core::domain_ (C++ member), 214
 hpx::resource::core::id (C++ function), 214

<code>hpx::resource::core::id_ (C++ member), 214</code>	<code>hpx::resource::pu::id_ (C++ member), 234</code>
<code>hpx::resource::core::invalid_core_id (C++ member), 214</code>	<code>hpx::resource::pu::invalid_pu_id (C++ member), 234</code>
<code>hpx::resource::core::pus (C++ function), 214</code>	<code>hpx::resource::pu::pu (C++ function), 234</code>
<code>hpx::resource::core::pus_ (C++ member), 214</code>	<code>hpx::resource::pu::pus_sharing_core (C++ function), 234</code>
<code>hpx::resource::get_partitioner (C++ function), 488</code>	<code>hpx::resource::pu::pus_sharing_numa_domain (C++ function), 234</code>
<code>hpx::resource::is_partitioner_valid (C++ function), 488</code>	<code>hpx::resource::pu::thread_occupancy_ (C++ member), 234</code>
<code>hpx::resource::local (C++ enumerator), 488</code>	<code>hpx::resource::pu::thread_occupancy_count_ (C++ member), 234</code>
<code>hpx::resource::local_priority_fifo (C++ enumerator), 488</code>	<code>hpx::resource::scheduler_function (C++ type), 487</code>
<code>hpx::resource::local_priority_lifo (C++ enumerator), 488</code>	<code>hpx::resource::scheduling_policy (C++ enum), 488</code>
<code>hpx::resource::mode_allow_dynamic_pools (C++ enumerator), 487</code>	<code>hpx::resource::shared_priority (C++ enumerator), 488</code>
<code>hpx::resource::mode_allow_oversubscription (C++ enumerator), 487</code>	<code>hpx::resource::static_ (C++ enumerator), 488</code>
<code>hpx::resource::mode_default (C++ enumerator), 487</code>	<code>hpx::resource::static_priority (C++ enumerator), 488</code>
<code>hpx::resource::numa_domain (C++ class), 224</code>	<code>hpx::resource::unspecified (C++ enumerator), 488</code>
<code>hpx::resource::numa_domain::cores (C++ function), 224</code>	<code>hpx::resource::user_defined (C++ enumerator), 488</code>
<code>hpx::resource::numa_domain::cores_ (C++ member), 224</code>	<code>hpx::resume (C++ function), 275</code>
<code>hpx::resource::numa_domain::id (C++ function), 224</code>	<code>hpx::rethrow (C++ enumerator), 253</code>
<code>hpx::resource::numa_domain::id_ (C++ member), 224</code>	<code>hpx::runtime_mode (C++ enum), 253</code>
<code>hpx::resource::numa_domain::invalid_numa_domain_id_ (C++ member), 224</code>	<code>hpx::runtime_mode_connect (C++ enumerator), 253</code>
<code>hpx::resource::numa_domain::numa_domain (C++ function), 224</code>	<code>hpx::runtime_mode_console (C++ enumerator), 253</code>
<code>hpx::resource::partitioner (C++ class), 232</code>	<code>hpx::runtime_mode_default (C++ enumerator), 253</code>
<code>hpx::resource::partitioner::add_resource (C++ function), 233</code>	<code>hpx::runtime_mode_invalid (C++ enumerator), 253</code>
<code>hpx::resource::partitioner::create_threadpool (C++ function), 232</code>	<code>hpx::runtime_mode_last (C++ enumerator), 253</code>
<code>hpx::resource::partitioner::get_default_policy (C++ function), 233</code>	<code>hpx::runtime_mode_worker (C++ enumerator), 253</code>
<code>hpx::resource::partitioner::numa_domainshpx::service_unavailable (C++ enumerator), 251</code>	<code>hpx::serialization_error (C++ enumerator), 252</code>
<code>hpx::resource::partitioner::partitioner (C++ function), 232</code>	<code>hpx::set_lco_error (C++ function), 302–304</code>
<code>hpx::resource::partitioner::partitioner_hpx::set_lco_value (C++ function), 299–301</code>	<code>hpx::set_lco_value_unmanaged (C++ function), 300, 301</code>
<code>hpx::resource::partitioner::set_default_policy (C++ function), 233</code>	<code>hpx::set_parcel_write_handler (C++ function), 297</code>
<code>hpx::resource::partitioner_mode (C++ enum), 487</code>	<code>hpx::shutdown_function_type (C++ type), 250</code>
<code>hpx::resource::pu (C++ class), 234</code>	<code>hpx::split_future (C++ function), 307</code>
<code>hpx::resource::pu::core_ (C++ member), 234</code>	<code>hpx::start (C++ function), 262–271</code>
<code>hpx::resource::pu::id (C++ function), 234</code>	<code>hpx::start_active_counters (C++ function), 284</code>

hpx::startup_function_type (C++ type), 250
 hpx::startup_timed_out (C++ enumerator), 251
 hpx::stop (C++ function), 274
 hpx::stop_active_counters (C++ function), 284
 hpx::success (C++ enumerator), 251
 hpx::suspend (C++ function), 275
 hpx::task_already_started (C++ enumerator), 252
 hpx::task_block_not_active (C++ enumerator), 253
 hpx::task_canceled_exception (C++ enumerator), 253
 hpx::task_moved (C++ enumerator), 252
 hpx::this_thread (C++ type), 488
 hpx::this_thread::get_executor (C++ function), 490
 hpx::this_thread::get_pool (C++ function), 491
 hpx::this_thread::suspend (C++ function), 488–490
 hpx::thread_cancelled (C++ enumerator), 252
 hpx::thread_interrupted (C++ class), 245
 hpx::thread_not_interruptable (C++ enumerator), 252
 hpx::thread_resource_error (C++ enumerator), 252
 hpx::threads (C++ type), 491
 hpx::threads::active (C++ enumerator), 491
 hpx::threads::depleted (C++ enumerator), 491
 hpx::threads::enumerate_threads (C++ function), 494
 hpx::threads::get_ctx_ptr (C++ function), 493
 hpx::threads::get_executor (C++ function), 499
 hpx::threads::get_numa_node_number (C++ function), 497
 hpx::threads::get_parent_id (C++ function), 493
 hpx::threads::get_parent_locality_id (C++ function), 494
 hpx::threads::get_parent_phase (C++ function), 494
 hpx::threads::get_pool (C++ function), 500
 hpx::threads::get_self (C++ function), 493
 hpx::threads::get_self_component_id (C++ function), 494
 hpx::threads::get_self_id (C++ function), 493
 hpx::threads::get_self_ptr (C++ function), 493
 hpx::threads::get_self_ptr_checked (C++ function), 493
 hpx::threads::get_self_stacksize (C++ function), 494
 hpx::threads::get_stack_size (C++ function), 499
 hpx::threads::get_stack_size_name (C++ function), 493
 hpx::threads::get_thread_count (C++ function), 494
 hpx::threads::get_thread_description (C++ function), 496
 hpx::threads::get_thread_interruption_enabled (C++ function), 498
 hpx::threads::get_thread_interruption_requested (C++ function), 498
 hpx::threads::get_thread_lco_description (C++ function), 497
 hpx::threads::get_thread_phase (C++ function), 497
 hpx::threads::get_thread_priority (C++ function), 499
 hpx::threads::get_thread_priority_name (C++ function), 493
 hpx::threads::get_thread_state (C++ function), 497
 hpx::threads::get_thread_state_ex_name (C++ function), 493
 hpx::threads::get_thread_state_name (C++ function), 493
 hpx::threads::interrupt_thread (C++ function), 498, 499
 hpx::threads::interruption_point (C++ function), 499
 hpx::threads::pending (C++ enumerator), 491
 hpx::threads::pending_boost (C++ enumerator), 491
 hpx::threads::pending_do_not_schedule (C++ enumerator), 491
 hpx::threads::policies (C++ type), 500
 hpx::threads::policies::default_mode (C++ enumerator), 501
 hpx::threads::policies::delay_exit (C++ enumerator), 500
 hpx::threads::policies::do_background_work (C++ enumerator), 500
 hpx::threads::policies::enable_elasticity (C++ enumerator), 500
 hpx::threads::policies::enable_idle_backoff (C++ enumerator), 501
 hpx::threads::policies::enable_stealing (C++ enumerator), 500
 hpx::threads::policies::nothing_special (C++ enumerator), 500
 hpx::threads::policies::reduce_thread_priority (C++ enumerator), 500

`hpx::threads::policies::scheduler_mode` (C++ *enum*), 500

`hpx::threads::set_thread_description` (C++ *function*), 496

`hpx::threads::set_thread_interruption_enabled` (C++ *function*), 498

`hpx::threads::set_thread_lco_description` (C++ *function*), 497

`hpx::threads::set_thread_state` (C++ *function*), 495, 496

`hpx::threads::staged` (C++ *enumerator*), 491

`hpx::threads::suspended` (C++ *enumerator*), 491

`hpx::threads::terminated` (C++ *enumerator*), 491

`hpx::threads::thread_pool_base` (C++ *class*), 246

`hpx::threads::thread_pool_base::resume` (C++ *function*), 246

`hpx::threads::thread_pool_base::resume_checkpoint` (C++ *function*), 246

`hpx::threads::thread_pool_base::resume_direct` (C++ *function*), 247

`hpx::threads::thread_pool_base::resume_processing_unit` (C++ *function*), 248

`hpx::threads::thread_pool_base::resume_processing_unit_thread` (C++ *function*), 248

`hpx::threads::thread_pool_base::suspend` (C++ *function*), 247

`hpx::threads::thread_pool_base::suspend_direct` (C++ *function*), 247

`hpx::threads::thread_pool_base::suspend_processing_unit` (C++ *function*), 247

`hpx::threads::thread_pool_base::suspend_processing_unit_thread` (C++ *function*), 248

`hpx::threads::thread_priority` (C++ *enum*), 491

`hpx::threads::thread_priority_boost` (C++ *enumerator*), 492

`hpx::threads::thread_priority_default` (C++ *enumerator*), 492

`hpx::threads::thread_priority_high` (C++ *enumerator*), 492

`hpx::threads::thread_priority_high_recursive` (C++ *enumerator*), 492

`hpx::threads::thread_priority_low` (C++ *enumerator*), 492

`hpx::threads::thread_priority_normal` (C++ *enumerator*), 492

`hpx::threads::thread_priority_unknown` (C++ *enumerator*), 491

`hpx::threads::thread_schedule_hint` (C++ *class*), 248

`hpx::threads::thread_schedule_hint::hint` (C++ *member*), 249

`hpx::threads::thread_schedule_hint::mode` (C++ *member*), 249

`hpx::threads::thread_schedule_hint::thread_scheduler` (C++ *function*), 249

`hpx::threads::thread_schedule_hint_mode` (C++ *enum*), 493

`hpx::threads::thread_schedule_hint_mode_none` (C++ *enumerator*), 493

`hpx::threads::thread_schedule_hint_mode_numa` (C++ *enumerator*), 493

`hpx::threads::thread_schedule_hint_mode_thread` (C++ *enumerator*), 493

`hpx::threads::thread_stacksize` (C++ *enum*), 492

`hpx::threads::thread_stacksize_current` (C++ *enumerator*), 492

`hpx::threads::thread_stacksize_default` (C++ *enumerator*), 493

`hpx::threads::thread_stacksize_huge` (C++ *enumerator*), 492

`hpx::threads::thread_stacksize_large` (C++ *enumerator*), 492

`hpx::threads::thread_stacksize_maximal` (C++ *enumerator*), 493

`hpx::threads::thread_stacksize_medium` (C++ *enumerator*), 492

`hpx::threads::thread_stacksize_minimal` (C++ *enumerator*), 493

`hpx::threads::thread_stacksize_small` (C++ *enumerator*), 492

`hpx::threads::thread_stacksize_unknown` (C++ *enumerator*), 492

`hpx::threads::thread_state_enum` (C++ *enum*), 491

`hpx::threads::thread_state_ex_enum` (C++ *enum*), 492

`hpx::threads::unknown` (C++ *enumerator*), 491

`hpx::threads::wait_abort` (C++ *enumerator*), 492

`hpx::threads::wait_signaled` (C++ *enumerator*), 492

`hpx::threads::wait_terminate` (C++ *enumerator*), 492

`hpx::threads::wait_timeout` (C++ *enumerator*), 492

`hpx::threads::wait_unknown` (C++ *enumerator*), 492

`hpx::throwmode` (C++ *enum*), 253

`hpx::throws` (C++ *member*), 322

`hpx::tolerate_node_faults` (C++ *function*), 283

hpx::traits (C++ type), 501
 hpx::trigger_lco_event (C++ function), 299
 hpx::unhandled_exception (C++ enumerator), 252
 hpx::uninitialized_value (C++ enumerator), 251
 hpx::unknown_component_address (C++ enumerator), 251
 hpx::unknown_error (C++ enumerator), 253
 hpx::unregister_thread (C++ function), 283
 hpx::unregister_with_basename (C++ function), 288
 hpx::util (C++ type), 501
 hpx::util::attach_debugger (C++ function), 505
 hpx::util::checkpoint (C++ class), 211
 hpx::util::checkpoint::~~checkpoint (C++ function), 211
 hpx::util::checkpoint::begin (C++ function), 211
 hpx::util::checkpoint::checkpoint (C++ function), 211
 hpx::util::checkpoint::const_iterator (C++ type), 211
 hpx::util::checkpoint::data (C++ member), 211
 hpx::util::checkpoint::end (C++ function), 211
 hpx::util::checkpoint::operator!= (C++ function), 211
 hpx::util::checkpoint::operator= (C++ function), 211
 hpx::util::checkpoint::operator== (C++ function), 211
 hpx::util::checkpoint::serialize (C++ function), 211
 hpx::util::checkpoint::size (C++ function), 211
 hpx::util::functional (C++ type), 509
 hpx::util::functional::invoke (C++ class), 222
 hpx::util::functional::invoke_r (C++ class), 222
 hpx::util::functional::unwrap (C++ class), 249
 hpx::util::functional::unwrap_all (C++ class), 249
 hpx::util::functional::unwrap_n (C++ class), 249
 hpx::util::operator>> (C++ function), 501
 hpx::util::operator<< (C++ function), 501
 hpx::util::restore_checkpoint (C++ function), 504
 hpx::util::save_checkpoint (C++ function), 501–504
 hpx::util::traverse_pack_async (C++ function), 506
 hpx::util::traverse_pack_async_allocator (C++ function), 507
 hpx::util::unwrap (C++ function), 508
 hpx::util::unwrap_all (C++ function), 509
 hpx::util::unwrap_n (C++ function), 509
 hpx::util::unwrapping (C++ function), 509
 hpx::util::unwrapping_all (C++ function), 509
 hpx::util::unwrapping_n (C++ function), 509
 hpx::version_too_new (C++ enumerator), 251
 hpx::version_too_old (C++ enumerator), 251
 hpx::version_unknown (C++ enumerator), 251
 hpx::wait_all (C++ function), 307, 308
 hpx::wait_all_n (C++ function), 308
 hpx::wait_any (C++ function), 310–312
 hpx::wait_any_n (C++ function), 312
 hpx::wait_each (C++ function), 319, 320
 hpx::wait_each_n (C++ function), 320
 hpx::wait_some (C++ function), 314, 315
 hpx::wait_some_n (C++ function), 316
 hpx::when_all (C++ function), 309
 hpx::when_all_n (C++ function), 310
 hpx::when_any (C++ function), 312, 313
 hpx::when_any_n (C++ function), 313
 hpx::when_any_result (C++ class), 249
 hpx::when_any_result::futures (C++ member), 249
 hpx::when_any_result::index (C++ member), 249
 hpx::when_each (C++ function), 320, 321
 hpx::when_each_n (C++ function), 321
 hpx::when_some (C++ function), 316–318
 hpx::when_some_n (C++ function), 318
 hpx::when_some_result (C++ class), 249
 hpx::when_some_result::futures (C++ member), 250
 hpx::when_some_result::indices (C++ member), 250
 hpx::yield_aborted (C++ enumerator), 252
 HPX_DECLARE_PLAIN_ACTION (C macro), 534
 HPX_DEFINE_COMPONENT_ACTION (C macro), 533
 HPX_DEFINE_PLAIN_ACTION (C macro), 534
 HPX_INVOKE (C macro), 541
 HPX_INVOKE_R (C macro), 541
 HPX_PLAIN_ACTION (C macro), 534
 HPX_PLAIN_ACTION_ID (C macro), 535
 HPX_REGISTER_ACTION (C macro), 532
 HPX_REGISTER_ACTION_DECLARATION (C macro), 531
 HPX_REGISTER_ACTION_ID (C macro), 532
 HPX_REGISTER_COMPONENT (C macro), 536

HPX_REGISTER_GATHER (*C macro*), 515
HPX_REGISTER_GATHER_DECLARATION (*C macro*), 515
HPX_SCHEDULER_MAX_TERMINATED_THREADS:STRING
 command line option, 76
HPX_THROW_EXCEPTION (*C macro*), 540
HPX_THROWS_IF (*C macro*), 540
HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL
 command line option, 77
HPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT:BOOL
 command line option, 72
HPX_WITH_APEX:BOOL
 command line option, 78
HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL
 command line option, 79
HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL
 command line option, 72
HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH
 command line option, 72
HPX_WITH_BUILD_BINARY_PACKAGE:BOOL
 command line option, 72
HPX_WITH_COMPILE_ONLY_TESTS:BOOL
 command line option, 74
HPX_WITH_COMPILER_WARNINGS:BOOL
 command line option, 72
HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL
 command line option, 72
HPX_WITH_COMPRESSION_BZIP2:BOOL
 command line option, 72
HPX_WITH_COMPRESSION_SNAPPY:BOOL
 command line option, 72
HPX_WITH_COMPRESSION_ZLIB:BOOL
 command line option, 72
HPX_WITH_CUDA:BOOL
 command line option, 72
HPX_WITH_CUDA_CLANG:BOOL
 command line option, 72
HPX_WITH_CXX14_RETURN_TYPE_DEDUCTION:BOOL
 command line option, 72
HPX_WITH_DATAPAR_BOOST_SIMD:BOOL
 command line option, 72
HPX_WITH_DATAPAR_VC:BOOL
 command line option, 72
HPX_WITH_DEFAULT_TARGETS:BOOL
 command line option, 74
HPX_WITH_DEPRECATED_WARNINGS:BOOL
 command line option, 72
HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL
 command line option, 72
HPX_WITH_DOCUMENTATION:BOOL
 command line option, 74
HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS:STRING
 command line option, 74
HPX_WITH_DYNAMIC_HPX_MAIN:BOOL
 command line option, 72
HPX_WITH_EXAMPLES:BOOL
 command line option, 74
HPX_WITH_EXAMPLES_HDF5:BOOL
 command line option, 74
HPX_WITH_EXAMPLES_OPENMP:BOOL
 command line option, 74
HPX_WITH_EXAMPLES_QT4:BOOL
 command line option, 74
HPX_WITH_EXAMPLES_QTHREADS:BOOL
 command line option, 75
HPX_WITH_EXAMPLES_TBB:BOOL
 command line option, 75
HPX_WITH_EXECUTABLE_PREFIX:STRING
 command line option, 75
HPX_WITH_FAIL_COMPILE_TESTS:BOOL
 command line option, 75
HPX_WITH_FAULT_TOLERANCE:BOOL
 command line option, 72
HPX_WITH_FORTRAN:BOOL
 command line option, 72
HPX_WITH_FULL_RPATH:BOOL
 command line option, 72
HPX_WITH_GCC_VERSION_CHECK:BOOL
 command line option, 73
HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL
 command line option, 73
HPX_WITH_GOOGLE_PERFTOOLS:BOOL
 command line option, 78
HPX_WITH_HCC:BOOL
 command line option, 73
HPX_WITH_HIDDEN_VISIBILITY:BOOL
 command line option, 73
HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY:BOOL
 command line option, 73
HPX_WITH_IO_COUNTERS:BOOL
 command line option, 75
HPX_WITH_IO_POOL:BOOL
 command line option, 76
HPX_WITH_ITTNOTIFY:BOOL
 command line option, 78
HPX_WITH_LOGGING:BOOL
 command line option, 73
HPX_WITH_MALLOC:STRING
 command line option, 73
HPX_WITH_MAX_CPU_COUNT:STRING
 command line option, 76
HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING
 command line option, 76
HPX_WITH_MORE_THAN_64_THREADS:BOOL
 command line option, 76
HPX_WITH_NATIVE_TLS:BOOL
 command line option, 73
HPX_WITH_NETWORKING:BOOL

command line option, 77	command line option, 79
HPX_WITH_NICE_THREADLEVEL:BOOL	HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING
command line option, 73	command line option, 79
HPX_WITH_PAPI:BOOL	HPX_WITH_TESTS_EXAMPLES:BOOL
command line option, 78	command line option, 75
HPX_WITH_PARCEL_COALESCING:BOOL	HPX_WITH_TESTS_EXTERNAL_BUILD:BOOL
command line option, 73	command line option, 75
HPX_WITH_PARCEL_PROFILING:BOOL	HPX_WITH_TESTS_HEADERS:BOOL
command line option, 78	command line option, 75
HPX_WITH_PARCELPOR_ACTION_COUNTERS:BOOL	HPX_WITH_TESTS_REGRESSIONS:BOOL
command line option, 77	command line option, 75
HPX_WITH_PARCELPOR_LIBFABRIC:BOOL	HPX_WITH_TESTS_UNIT:BOOL
command line option, 78	command line option, 75
HPX_WITH_PARCELPOR_MPI:BOOL	HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING
command line option, 78	command line option, 76
HPX_WITH_PARCELPOR_MPI_ENV:STRING	HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL
command line option, 78	command line option, 76
HPX_WITH_PARCELPOR_MPI_MULTITHREADED:BOOL	HPX_WITH_THREAD_COMPATIBILITY:BOOL
command line option, 78	command line option, 73
HPX_WITH_PARCELPOR_TCP:BOOL	HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL
command line option, 78	command line option, 76
HPX_WITH_PARCELPOR_VERBS:BOOL	HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL
command line option, 78	command line option, 76
HPX_WITH_PSEUDO_DEPENDENCIES:BOOL	HPX_WITH_THREAD_DEBUG_INFO:BOOL
command line option, 75	command line option, 79
HPX_WITH_QUEUE_COMPATIBILITY:BOOL	HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL
command line option, 73	command line option, 79
HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL	HPX_WITH_THREAD_GUARD_PAGE:BOOL
command line option, 73	command line option, 79
HPX_WITH_SANITIZERS:BOOL	HPX_WITH_THREAD_IDLE_RATES:BOOL
command line option, 79	command line option, 77
HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL	HPX_WITH_THREAD_LOCAL_STORAGE:BOOL
command line option, 76	command line option, 77
HPX_WITH_SCOPED_UNLOCK_COMPATIBILITY:BOOL	HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL
command line option, 73	command line option, 77
HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL	HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL
command line option, 76	command line option, 77
HPX_WITH_SPINLOCK_POOL_NUM:STRING	HPX_WITH_THREAD_SCHEDULERS:STRING
command line option, 76	command line option, 77
HPX_WITH_STACKOVERFLOW_DETECTION:BOOL	HPX_WITH_THREAD_STACK_MMAP:BOOL
command line option, 73	command line option, 77
HPX_WITH_STACKTRACES:BOOL	HPX_WITH_THREAD_STEALING_COUNTS:BOOL
command line option, 76	command line option, 77
HPX_WITH_STATIC_LINKING:BOOL	HPX_WITH_THREAD_TARGET_ADDRESS:BOOL
command line option, 73	command line option, 77
HPX_WITH_SWAP_CONTEXT_EMULATION:BOOL	HPX_WITH_TIMER_POOL:BOOL
command line option, 76	command line option, 77
HPX_WITH_SYCL:BOOL	HPX_WITH_TOOLS:BOOL
command line option, 73	command line option, 75
HPX_WITH_TESTS:BOOL	HPX_WITH_UNWRAPPED_COMPATIBILITY:BOOL
command line option, 75	command line option, 73
HPX_WITH_TESTS_BENCHMARKS:BOOL	HPX_WITH_VALGRIND:BOOL
command line option, 75	command line option, 79
HPX_WITH_TESTS_DEBUG_LOG:BOOL	HPX_WITH_VERIFY_LOCKS:BOOL

command line option, [79](#)
HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL
command line option, [79](#)
HPX_WITH_VERIFY_LOCKS_GLOBALLY:BOOL
command line option, [79](#)
HPX_WITH_VIM_YCM:BOOL
command line option, [73](#)
HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING
command line option, [73](#)
HWLOC_ROOT:PATH
command line option, [79](#)

L

LCO, [18](#)
Lightweight Control Object, [18](#)
Local Control Object, [18](#)
Locality, [17](#)

M

make_error_code (C++ *function*), [220](#)

O

operator>> (C++ *function*), [212](#)
operator<< (C++ *function*), [212](#)

P

PAPI_ROOT:PATH
command line option, [80](#)
Parcel, [17](#)
Process, [17](#)

R

restore_checkpoint (C++ *function*), [212](#)

S

SPHINX_ROOT:PATH
command line option, [45](#)