
HPX Documentation

master

The STE||AR Group

January 18, 2022

USER DOCUMENTATION

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

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

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

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

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

² <https://cpplang.slack.com>

³ hpx-users@stellar.cct.lsu.edu

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

**CHAPTER
ONE**

WHAT IS HPX?

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

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

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

WHAT'S SO SPECIAL ABOUT *HPX*?

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

2.1 Quick start

The following steps will help you get started with *HPX*.

2.1.1 Installing *HPX*

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

1. **vcpkg**

You can download and install *HPX* using the `vcpkg`⁵ dependency manager:

```
$ vcpkg install hpx
```

2. **Spack**

Another way to install *HPX* is using `Spack`⁶:

```
$ spack install hpx
```

⁵ <https://github.com/Microsoft/vcpkg>

⁶ <https://spack.readthedocs.io/en/latest/>

3. Fedora

Installation can be done with Fedora⁷ as well:

```
$ dnf install hpx*
```

4. Arch Linux

HPX is available in the [Arch User Repository \(AUR\)](#)⁸ as hpx too.

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

2.1.2 Hello, World!

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

```
cmake_minimum_required(VERSION 3.18)
project(my_hpx_project CXX)
find_package(HPX REQUIRED)
add_executable(my_hpx_program main.cpp)
target_link_libraries(my_hpx_program HPX::hpx HPX::wrap_main HPX::iostreams_component)
```

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

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

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

Then, in your project directory run the following:

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

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

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

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

⁷ <https://fedoraproject.org/wiki/DNF>

⁸ https://wiki.archlinux.org/title/Arch_User_Repository

Note:

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

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

2.1.3 Writing task-based applications

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

```

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

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

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

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

2.1.4 Next steps

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

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

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

2.2 Examples

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

2.2.1 Asynchronous execution

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

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

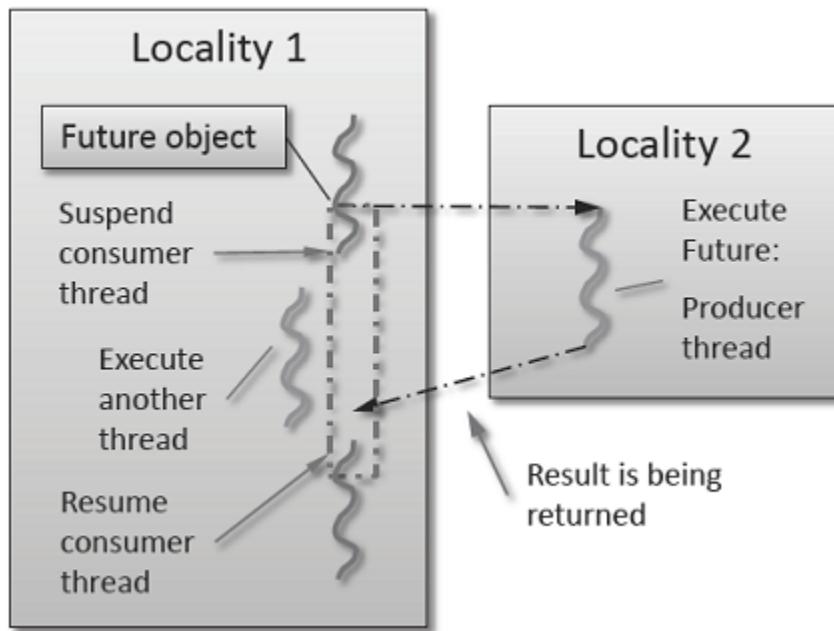


Fig. 2.1: Schematic of a future execution.

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

Setup

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

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

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

To run the program type:

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

This should print (time should be approximate):

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

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

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

Will yield:

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

Walkthrough

Now that you have compiled and run the code, let's look at how the code works. Since this code is written in C++, we will begin with the `main()` function. Here you can see that in *HPX*, `main()` is only used to initialize the runtime system. It is important to note that application-specific command line options are defined here. *HPX* uses [Boost.Program Options](#)⁹ for command line processing. You can see that our programs `--n-value` option is set by calling the `add_options()` method on an instance of `hpx::program_options::options_description`. The default value of the variable is set to 10. This is why when we ran the program for the first time without using the `--n-value` option the program returned the 10th value of the Fibonacci sequence. The constructor argument of the description is the text that appears when a user uses the `--hpx:help` option to see what command line options are available. `HPX_APPLICATION_STRING` is a macro that expands to a string constant containing the name of the *HPX* application currently being compiled.

In *HPX* `main()` is used to initialize the runtime system and pass the command line arguments to the program. If you wish to add command line options to your program you would add them here using the instance of the Boost class `options_description`, and invoking the public member function `.add_options()` (see [Boost Documentation](#)¹⁰ for more details). `hpx::init` calls `hpx_main()` after setting up *HPX*, which is where the logic of our program is encoded.

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

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

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

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

⁹ https://www.boost.org/doc/html/program_options.html

¹⁰ <https://www.boost.org/doc/>

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

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

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

        std::uint64_t r = fibonacci(n);

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

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

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

```
std::uint64_t fibonacci(std::uint64_t n)
{
    if (n < 2)
        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* presents dataflow, which is a way to more efficiently chain together multiple tasks.

2.2.2 Parallel algorithms

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

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

Matrix B is :
4 9
6 1
9 8

Resultant Matrix is :
124 93
111 127
```

Setup

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

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

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

To run the program type:

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

or:

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

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

This should print:

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

Matrix B is :
4 9
6 1
9 8

Resultant Matrix is :
124 93
111 127
```

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

Walkthrough

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

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

```
int main(int argc, char* argv[])
{
    using namespace hpx::program_options;
    options_description cmdline("usage: " HPX_APPLICATION_STRING " [options]");
    // clang-format off
    cmdline.add_options()
        ("n",
         hpx::program_options::value<std::size_t>()>default_value(2),
         "Number of rows of first matrix")
        ("m",
         hpx::program_options::value<std::size_t>()>default_value(3),
         "Number of columns of first matrix (equal to the number of rows of "
         "second matrix)")
        ("k",
         hpx::program_options::value<std::size_t>()>default_value(2),
         "Number of columns of second matrix")
        ("seed,s",
         hpx::program_options::value<unsigned int>(),
         "The random number generator seed to use for this run")
        ("l",
         hpx::program_options::value<std::size_t>()>default_value(0),
         "Lower limit of range of values")
        ("u",
         hpx::program_options::value<std::size_t>()>default_value(10),
         "Upper limit of range of values");
    // clang-format on
    hpx::local::init_params init_args;
    init_args.desc_cmdline = cmdline;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

// Define range of values
std::size_t lower = vm["l"].as<std::size_t>();
std::size_t upper = vm["u"].as<std::size_t>();

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

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

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

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

```

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

To give some random initial values to our matrices, we use `std::uniform_int_distribution`¹¹. Then, `std::bind()` is used along with `hpx::ranges::generate()` to yield two matrices A and B, which contain values in the range of $[0, 10]$ or in the range defined by the user at the command-line arguments. The seed to generate the values can also be defined by the user.

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

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

```

void print_matrix(
    std::vector<int> M, std::size_t rows, std::size_t cols, const char* message)
{

```

(continues on next page)

¹¹ https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

(continued from previous page)

```
std::cout << "\nMatrix " << message << " is:" << std::endl;
for (std::size_t i = 0; i < rows; i++)
{
    for (std::size_t j = 0; j < cols; j++)
        std::cout << M[i * cols + j] << " ";
    std::cout << "\n";
}
}
```

2.2.3 Asynchronous execution with actions

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

Setup

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

To compile this program, go to your *HPX* build directory (see *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
    hpx::program_options::options_description
        desc_commandline("Usage: " HPX_APPLICATION_STRING " [options]");
    desc_commandline.add_options()
        ("n-value",
            hpx::program_options::value<std::uint64_t>() -> default_value(10),
            "n value for the Fibonacci function")
        ;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.4 Remote execution with actions

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

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

Setup

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

To compile this program, go to your *HPX* build directory (see *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::future<void>> futures;
    futures.reserve(localities.size());
```

(continues on next page)

(continued from previous page)

```

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

```

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

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

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

```

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

```

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

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

```

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

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

    // As long as there are still elements in the set, we must keep scheduling
    // HPX-threads. Because HPX features work-stealing task schedulers, we have
    // no way of enforcing which worker OS-thread will actually execute
    // each HPX-thread.
    while (!attendance.empty())
    {
        // Each iteration, we create a task for each element in the set of

```

(continues on next page)

(continued from previous page)

```

// OS-threads that have not said "Hello world". Each of these tasks
// is encapsulated in a future.
std::vector<hpx::future<std::size_t>> futures;
futures.reserve(attendance.size());

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

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

```

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

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

```

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

```

(continues on next page)

(continued from previous page)

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

    return desired;
}

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

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

2.2.5 Components and actions

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

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

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

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

The component in this example exposes three different functions:

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

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

Setup

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

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

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

To run the program type:

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

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

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

Walkthrough

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

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

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

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

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

The server class

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

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

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

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

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

```
argument_type value_;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

HPX_REGISTER_COMPONENT(accumulator_type, accumulator)

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

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

The client class

The following code is from `accumulator.hpp`.

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

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

and to invoke component actions:

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

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

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

For readability, we `typedef` the base class like so:

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

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

There are a few different ways of invoking actions:

- **Non-blocking:** For actions that don't have return types, or when we do not care about the result of an action, we can invoke the action using fire-and-forget semantics. This means that once we have asked HPX to compute the action, we forget about it completely and continue with our computation. We use `hpx::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*, *Asynchronous execution with actions*, and the *Remote execution with actions*, enable asynchronous action invocation. Here's an example from the `accumulator` client class:

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

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

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

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

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

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

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

2.2.6 Dataflow

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

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

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

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

$$I = Pi$$

and

$$P = P + I$$

Setup

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

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

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

To run the program type:

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

Walkthrough

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

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

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

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

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

Next we look at hpx_main.

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

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

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

return hpx::finalize();
}

```

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

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

```

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

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

```

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

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

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

2.2.7 Local to remote

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

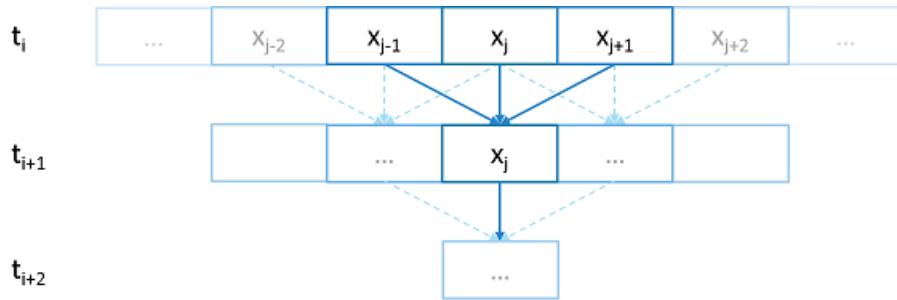


Fig. 2.2: Heat diffusion example program flow.

We parallelize this code over the following eight examples:

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

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

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

    // Our data for one time step
    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)
```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

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

```

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

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

```

(continues on next page)

(continued from previous page)

```

{
    using hpx::dataflow;
    using hpx::unwrapping;

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

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

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

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

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

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

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

```

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

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

By setting up the algorithm this way, the program will be able to execute as quickly as the dependencies of each future are met. Unfortunately, this example runs terribly slow. This increase in execution time is caused by the overheads

needed to create a future for each data point. Because the work done within each call to heat is very small, the overhead of creating and scheduling each of the three futures is greater than that of the actual useful work! In order to amortize the overheads of our synchronization techniques, we need to be able to control the amount of work that will be done with each future. We call this amount of work per overhead grain size.

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

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

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

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

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

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

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

```
for (std::size_t i = 0; i != np; ++i)
    U[0][i] = partition(localities[locidx(i, np, n1)], nx, double(i));
```

We distribute the partitions evenly based on the number of localities used, which is described in the function `locidx`. Because some of the data needed to update the partition in `heat_part` could now be on a new *locality*, we must

devise a way of moving data to the *locality* of the middle partition. We accomplished this by adding a switch in the function `get_data()` that returns the end element of the buffer `data_` if it is from the left partition or the first element of the buffer if the data is from the right partition. In this way only the necessary elements, not the whole buffer, are exchanged between nodes. The reader should be reminded that this exchange of end elements occurs in the function `get_data()` and, therefore, is executed asynchronously.

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

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

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

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

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

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

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

                // The new partition_data will be allocated on the same locality
                // as 'middle'.
                return partition(middle.get_id(), std::move(next));
            }
        )
    );
}
```

(continues on next page)

(continued from previous page)

```

        }
    ),
    std::move(next_middle),
    left.get_data(partition_server::left_partition),
    middle_data,
    right.get_data(partition_server::right_partition)
);
}

```

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

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

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

2.3 Manual

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

2.3.1 HPX build system

Prerequisites

Supported platforms

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

Table 2.1: Supported Platforms for *HPX*

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

Supported compilers

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

Table 2.2: Supported Compilers for *HPX*

Name	Minimum Version
GNU Compiler Collection (g++) ¹²	7.0
clang: a C language family frontend for LLVM ¹³	7.0
Visual C++ ¹⁴ (x64)	2015

Software and libraries

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

Table 2.3: Software prerequisites for *HPX*

	Name	Minimum Version
Build System	CMake ¹⁵	3.18
Required Libraries	Boost ¹⁶	1.71.0
	Portable Hardware Locality (HWLOC) ¹⁷	1.5
	Asio ¹⁸	1.12.0

The most important dependencies are Boost¹⁹ and Portable Hardware Locality (HWLOC)²⁰. The installation of Boost is described in detail in Boost's [Getting Started²¹](#) document. A recent version of hwloc is required in order to support thread pinning and NUMA awareness and can be found in [Hwloc Downloads²²](#).

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

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

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

¹² <https://gcc.gnu.org>

¹³ <https://clang.llvm.org/>

¹⁴ <https://msdn.microsoft.com/en-us/visualc/default.aspx>

¹⁵ <https://www.cmake.org>

¹⁶ <https://www.boost.org/>

¹⁷ <https://www.open-mpi.org/projects/hwloc/>

¹⁸ <https://think-async.com/asio/>

¹⁹ <https://www.boost.org/>

²⁰ <https://www.open-mpi.org/projects/hwloc/>

²¹ https://www.boost.org/more/getting_started/index.html

²² <https://www.open-mpi.org/software/hwloc/v1.11>

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

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

Table 2.4: Optional software prerequisites for *HPX*

Name	Minimum version
<code>google-perfetto</code> ²³	1.7.1
<code>jemalloc</code> ²⁴	2.1.0
<code>mi-malloc</code> ²⁵	1.0.0
Performance Application Programming Interface (PAPI)	

Getting *HPX*

Download a tarball of the latest release from *HPX* Downloads²⁶ and unpack it or clone the repository directly using git:

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

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

```
$ cd hpx
$ git checkout 1.7.1
```

Building *HPX*

Basic information

The build system for *HPX* is based on `CMake`²⁷, a cross-platform build-generator tool which is not responsible for building the project but rather generates the files needed by your build tool (GNU make, Visual Studio, etc.) for building *HPX*. If `CMake` is not already installed in your system, you can download it and install it here: `CMake Downloads`²⁸.

Once `CMake` has been run, the build process can be started. The *HPX* build process is highly configurable through `CMake`, and various `CMake` variables influence the build process. The build process consists of the following parts:

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

²³ <https://code.google.com/p/gperftools>

²⁴ <http://jemalloc.net>

²⁵ <http://microsoft.github.io/mimalloc/>

²⁶ <https://hpx.stellar-group.org/downloads/>

²⁷ <https://www.cmake.org>

²⁸ <https://www.cmake.org/cmake/resources/software.html>

- *HPX Documentation* (target `docs`): This target builds the documentation, and is not enabled by default (enable by setting `HPX_WITH_DOCUMENTATION:BOOL=On`). For more information see *Documentation*.

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

The variables can be used to refine the recipes that can be found at *Platform specific build recipes* which show some basic steps on how to build *HPX* for a specific platform.

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

Most important CMake options

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

HPX_WITH_MALLOC

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

HPX_WITH_CUDA

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

HPX_WITH_PARCELPORT_MPI

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

HPX_WITH_PARCELPORT_TCP

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

HPX_WITH_APEX

Enable APEX integration. [APEX](#)²⁹ can be used to profile *HPX* applications. In particular, it provides information about individual tasks in the *HPX* runtime.

HPX_WITH_GENERIC_CONTEXT_COROUTINES

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

HPX_WITH_MAX_CPU_COUNT

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

HPX_WITH_CXX_STANDARD

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

HPX_WITH_EXAMPLES

Build examples.

HPX_WITH_TESTS

Build tests.

²⁹ <https://uo-oaciss.github.io/apex/quickstarthpx/>

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](#)³⁰). Available build types are:

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

Important: We currently don't guarantee ABI compatibility between Debug and Release builds. Please make sure that applications built against *HPX* use the same build type as you used to build *HPX*. For CMake builds, this means that the `CMAKE_BUILD_TYPE` variables have to match and for projects not using [CMake](#)³¹, the `HPX_DEBUG` macro has to be set in debug mode.

Platform specific build recipes

Unix variants

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

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

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

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

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

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

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

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

```
$ cmake ..
```

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

³⁰ https://cmake.org/cmake/help/latest/variable/CMAKE_BUILD_TYPE.html

³¹ <https://www.cmake.org>

```
$ cmake -DBOOST_ROOT=/path/to/boost
      -DHWLOC_ROOT=/path/to/hwloc
      -DTCMALLOC_ROOT=/path/to/tcmalloc
      -DJEMALLOC_ROOT=/path/to/jemalloc
      [other CMake variable definitions]
      /path/to/source/tree
```

For instance:

```
$ cmake -DBOOST_ROOT=~/packages/boost -DHWLOC_ROOT=/packages/hwloc -
      -DCMAKE_INSTALL_PREFIX=~/packages/hpx ~/downloads/hpx_1.5.1
```

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

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

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

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

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

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

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

Windows

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

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

- Download the CMake V3.18.1 installer (or latest version) from [here³²](#)
- Download the hwloc V1.11.0 (or the latest version) from [here³³](#) and unpack it.
- Download the latest Boost libraries from [here³⁴](#) 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:

³² <https://blog.kitware.com/cmake-3-18-1-available-for-download/>

³³ <http://www.open-mpi.org/software/hwloc/v1.11/downloads/hwloc-win64-build-1.11.0.zip>

³⁴ <https://www.boost.org/users/download/>

```
bootstrap.bat
```

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

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

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

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

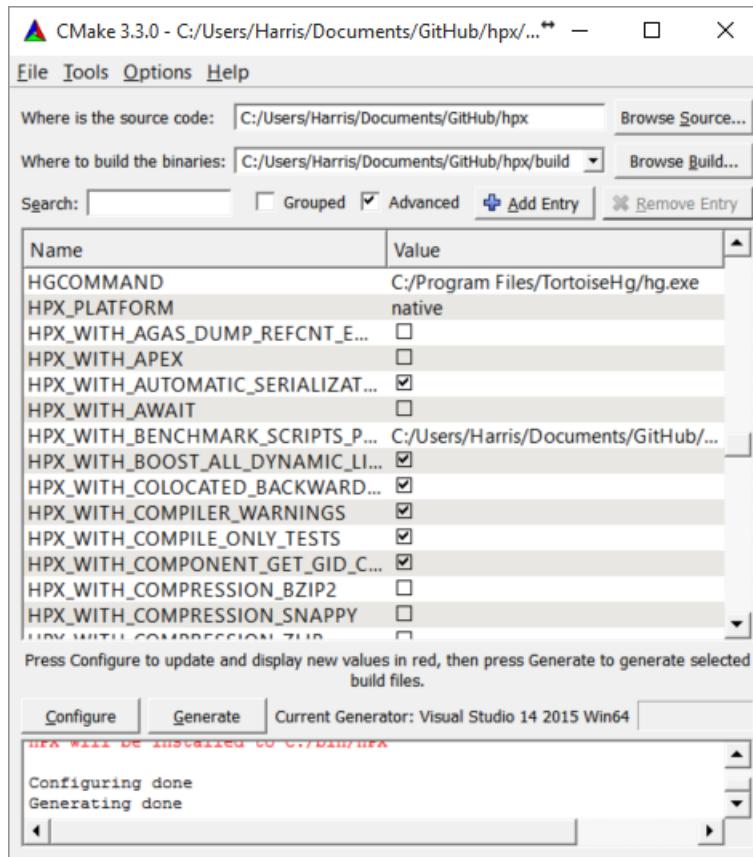


Fig. 2.3: Example CMake path settings.

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

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

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

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

This is how variable insertion will look:

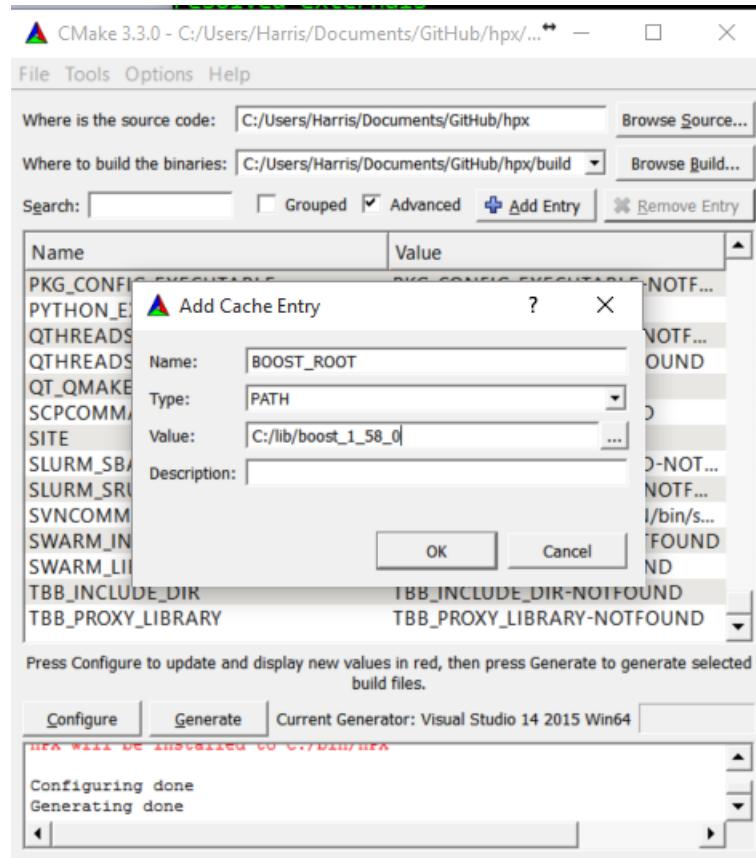


Fig. 2.4: Example CMake adding entry.

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

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

- Go to CMakePredefinedTargets and build the INSTALL project:

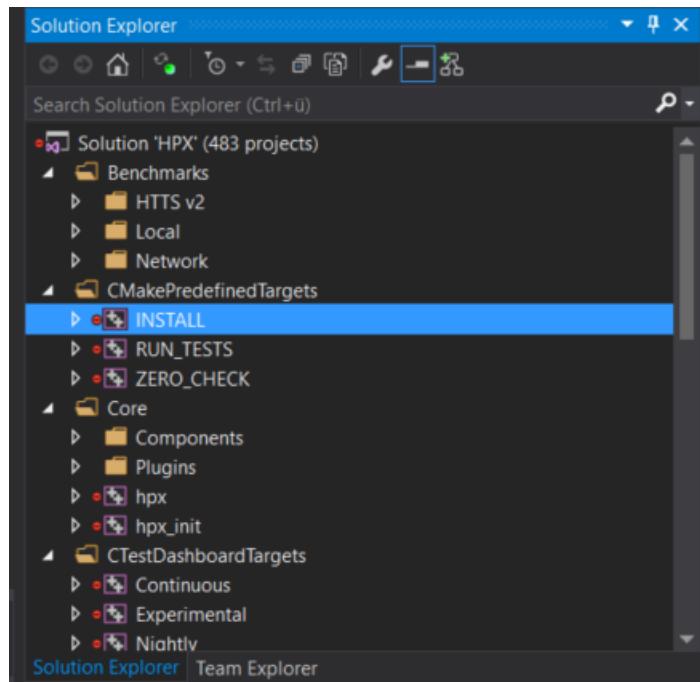


Fig. 2.5: Visual Studio INSTALL target.

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

Tests and examples

Running tests

To build the tests:

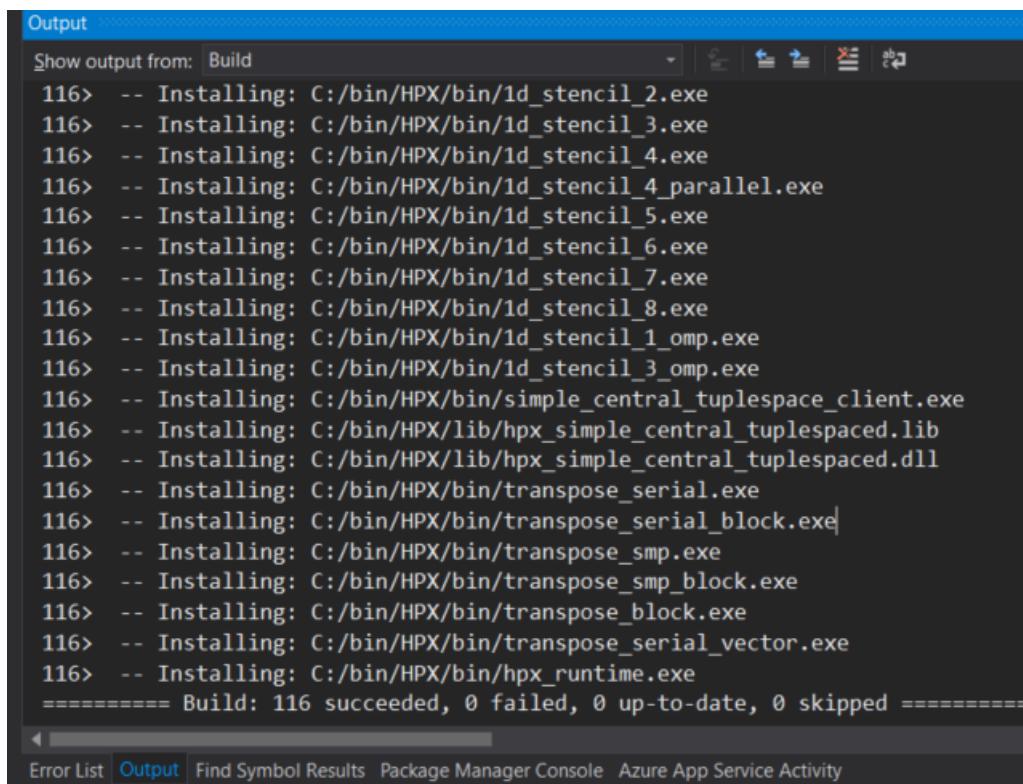
```
$ cmake --build . --target tests
```

To control which tests to run use ctest:

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

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

- To run a whole group of tests:



The screenshot shows the Visual Studio Output window with the title 'Output' at the top. The window displays a list of build logs:

```

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 =====

```

At the bottom of the window, there are tabs for Error List, Output (which is selected), Find Symbol Results, Package Manager Console, and Azure App Service Activity.

Fig. 2.6: Visual Studio build output.

```
$ ctest --output-on-failure -R tests.unit
```

Running examples

- To build (and install) all examples invoke:

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

- To build the `hello_world_1` example run:

```
$ make hello_world_1
```

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

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

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

Tip: Most build targets in *HPX* have two names: a simple name and a hierarchical name corresponding to what type of example or test the target is. If you are developing *HPX* it is often helpful to run `make help` to get a list of 23 build targets. For example, `make help | grep hello_world` outputs the following:

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

Variables that influence how HPX is built

The options are split into these categories:

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

Generic options

- `HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL`
- `HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH`
- `HPX_WITH_BUILD_BINARY_PACKAGE:BOOL`
- `HPX_WITH_CHECK_MODULE_DEPENDENCIES:BOOL`
- `HPX_WITH_COMPILER_WARNINGS:BOOL`
- `HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL`
- `HPX_WITH_COMPRESSION_BZIP2:BOOL`
- `HPX_WITH_COMPRESSION_SNAPPY:BOOL`
- `HPX_WITH_COMPRESSION_ZLIB:BOOL`
- `HPX_WITH_CUDA:BOOL`
- `HPX_WITH_CXX_STANDARD:STRING`
- `HPX_WITH_DATAPAR:BOOL`
- `HPX_WITH_DATAPAR_VC:BOOL`
- `HPX_WITH_DATAPAR_VC_NO_LIBRARY:BOOL`
- `HPX_WITH_DEPRECATED_WARNINGS:BOOL`
- `HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL`
- `HPX_WITH_DYNAMIC_HPX_MAIN:BOOL`
- `HPX_WITHFAULT_TOLERANCE:BOOL`
- `HPX_WITH_FULL_RPATH:BOOL`
- `HPX_WITH_GCC_VERSION_CHECK:BOOL`
- `HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL`
- `HPX_WITH_HIDDEN_VISIBILITY:BOOL`
- `HPX_WITH_HIP:BOOL`

- *HPX_WITH_LOGGING:BOOL*
- *HPX_WITH_MALLOC:STRING*
- *HPX_WITH_MODULES_AS_STATIC_LIBRARIES:BOOL*
- *HPX_WITH_NICE_THREADLEVEL:BOOL*
- *HPX_WITH_PARCEL_COALESCING:BOOL*
- *HPX_WITH_PKGCONFIG:BOOL*
- *HPX_WITH_PRECOMPILED_HEADERS:BOOL*
- *HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL*
- *HPX_WITH_STACKOVERFLOW_DETECTION:BOOL*
- *HPX_WITH_STATIC_LINKING:BOOL*
- *HPX_WITH_UNITY_BUILD:BOOL*
- *HPX_WITH_VIM_YCM:BOOL*
- *HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING*

HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL

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

HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH

Directory to place batch scripts in

HPX_WITH_BUILD_BINARY_PACKAGE:BOOL

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

HPX_WITH_CHECK_MODULE_DEPENDENCIES:BOOL

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

HPX_WITH_COMPILER_WARNINGS:BOOL

Enable compiler warnings (default: ON)

HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL

Turn compiler warnings into errors (default: OFF)

HPX_WITH_COMPRESSION_BZIP2:BOOL

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

HPX_WITH_COMPRESSION_SNAPPY:BOOL

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

HPX_WITH_COMPRESSION_ZLIB:BOOL

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

HPX_WITH_CUDA:BOOL

Enable support for CUDA (default: OFF)

HPX_WITH_CXX_STANDARD:STRING

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

HPX_WITH_DATAPAR:BOOL

Enable data parallel algorithm support (default: ON)

HPX_WITH_DATAPAR_VC:BOOL

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

HPX_WITH_DATAPAR_VC_NO_LIBRARY:BOOL

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

HPX_WITH_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 and Apple only, default: ON)

HPX_WITH_FAULT_TOLERANCE:BOOL

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

HPX_WITH_FULL_RPATH:BOOL

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

HPX_WITH_GCC_VERSION_CHECK:BOOL

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

HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL

Use Boost.Context as the underlying coroutines context switch implementation.

HPX_WITH_HIDDEN_VISIBILITY:BOOL

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

HPX_WITH_HIP:BOOL

Enable compilation with HIPCC (default: OFF)

HPX_WITH_LOGGING:BOOL

Build HPX with logging enabled (default: ON).

HPX_WITH_MALLOC:STRING

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

HPX_WITH_MODULES_AS_STATIC_LIBRARIES:BOOL

Compile HPX modules as static (whole-archive) libraries instead of OBJECT libraries (Default: OFF)

HPX_WITH_NICE_THREADLEVEL:BOOL

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

HPX_WITH_PARCEL_COALESCING:BOOL

Enable the parcel coalescing plugin (default: ON).

HPX_WITH_PKGCONFIG:BOOL

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

HPX_WITH_PRECOMPILED_HEADERS:BOOL

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

HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL

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

HPX_WITH_STACKOVERFLOW_DETECTION:BOOL

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

HPX_WITH_STATIC_LINKING:BOOL

Compile HPX statically linked libraries (Default: OFF)

HPX_WITH_UNITY_BUILD:BOOL

Enable unity build for certain build targets (default OFF)

HPX_WITH_VIM_YCM:BOOL

Generate HPX completion file for VIM YouCompleteMe plugin

HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING

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

Build Targets options

- *HPX_WITH_ASIO_TAG:STRING*
- *HPX_WITH_COMPILE_ONLY_TESTS:BOOL*
- *HPX_WITH_DISTRIBUTED_RUNTIME:BOOL*
- *HPX_WITH_DOCUMENTATION:BOOL*
- *HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS:STRING*
- *HPX_WITH_EXAMPLES:BOOL*
- *HPX_WITH_EXAMPLES_HDF5:BOOL*
- *HPX_WITH_EXAMPLES_OPENMP:BOOL*
- *HPX_WITH_EXAMPLES_QT4:BOOL*
- *HPX_WITH_EXAMPLES_QTHREADS:BOOL*
- *HPX_WITH_EXAMPLES_TBB:BOOL*
- *HPX_WITH_EXECUTABLE_PREFIX:STRING*
- *HPX_WITH_FAIL_COMPILE_TESTS:BOOL*
- *HPX_WITH_FETCH_ASIO:BOOL*
- *HPX_WITH_IO_COUNTERS:BOOL*
- *HPX_WITH_TESTS:BOOL*
- *HPX_WITH_TESTS_BENCHMARKS:BOOL*
- *HPX_WITH_TESTS_EXAMPLES:BOOL*
- *HPX_WITH_TESTS_EXTERNAL_BUILD:BOOL*
- *HPX_WITH_TESTS_HEADERS:BOOL*
- *HPX_WITH_TESTS_REGRESSIONS:BOOL*
- *HPX_WITH_TESTS_UNIT:BOOL*
- *HPX_WITH_TOOLS:BOOL*

HPX_WITH_ASIO_TAG:STRING

Asio repository tag or branch

HPX_WITH_COMPILE_ONLY_TESTS:BOOL

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

HPX_WITH_DISTRIBUTED_RUNTIME:BOOL

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

HPX_WITH_DOCUMENTATION:BOOL

Build the HPX documentation (default OFF).

HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS : STRING

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

HPX_WITH_EXAMPLES : BOOL

Build the HPX examples (default ON)

HPX_WITH_EXAMPLES_HDF5 : BOOL

Enable examples requiring HDF5 support (default: OFF).

HPX_WITH_EXAMPLES_OPENMP : BOOL

Enable examples requiring OpenMP support (default: OFF).

HPX_WITH_EXAMPLES_QT4 : BOOL

Enable examples requiring Qt4 support (default: OFF).

HPX_WITH_EXAMPLES_QTHREADS : BOOL

Enable examples requiring QThreads support (default: OFF).

HPX_WITH_EXAMPLES_TBB : BOOL

Enable examples requiring TBB support (default: OFF).

HPX_WITH_EXECUTABLE_PREFIX : STRING

Executable prefix (default none), ‘**hpx**’ useful for system install.

HPX_WITH_FAIL_COMPILE_TESTS : BOOL

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

HPX_WITH_FETCH_ASIO : BOOL

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

HPX_WITH_IO_COUNTERS : BOOL

Enable IO counters (default: ON)

HPX_WITH_TESTS : BOOL

Build the HPX tests (default ON)

HPX_WITH_TESTS_BENCHMARKS : BOOL

Build HPX benchmark tests (default: ON)

HPX_WITH_TESTS_EXAMPLES : BOOL

Add HPX examples as tests (default: ON)

HPX_WITH_TESTS_EXTERNAL_BUILD : BOOL

Build external cmake build tests (default: ON)

HPX_WITH_TESTS_HEADERS : BOOL

Build HPX header tests (default: OFF)

HPX_WITH_TESTS_REGRESSIONS : BOOL

Build HPX regression tests (default: ON)

HPX_WITH_TESTS_UNIT : BOOL

Build HPX unit tests (default: ON)

HPX_WITH_TOOLS : BOOL

Build HPX tools (default: OFF)

Thread Manager options

- `HPX_COROUTINES_WITH_SWAP_CONTEXT_EMULATION:BOOL`
- `HPX_WITH_COROUTINE_COUNTERS:BOOL`
- `HPX_WITH_IO_POOL:BOOL`
- `HPX_WITH_MAX_CPU_COUNT:STRING`
- `HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING`
- `HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL`
- `HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL`
- `HPX_WITH_SPINLOCK_POOL_NUM:STRING`
- `HPX_WITH_STACKTRACES:BOOL`
- `HPX_WITH_STACKTRACES_DEMANGLE_SYMBOLS:BOOL`
- `HPX_WITH_STACKTRACES_STATIC_SYMBOLS:BOOL`
- `HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING`
- `HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL`
- `HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL`
- `HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL`
- `HPX_WITH_THREAD_IDLE_RATES:BOOL`
- `HPX_WITH_THREAD_LOCAL_STORAGE:BOOL`
- `HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL`
- `HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL`
- `HPX_WITH_THREAD_STACK_MMAP:BOOL`
- `HPX_WITH_THREAD_STEALING_COUNTS:BOOL`
- `HPX_WITH_THREAD_TARGET_ADDRESS:BOOL`
- `HPX_WITH_TIMER_POOL:BOOL`

`HPX_COROUTINES_WITH_SWAP_CONTEXT_EMULATION:BOOL`

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

`HPX_WITH_COROUTINE_COUNTERS:BOOL`

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

`HPX_WITH_IO_POOL:BOOL`

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

`HPX_WITH_MAX_CPU_COUNT:STRING`

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

`HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING`

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

`HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL`

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

`HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL`

Enable spinlock deadlock detection (default: OFF)

HPX_WITH_SPINLOCK_POOL_NUM: STRING

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

HPX_WITH_STACKTRACES: BOOL

Attach backtraces to HPX exceptions (default: ON)

HPX_WITH_STACKTRACES_DEMANGLE_SYMBOLS: BOOL

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

HPX_WITH_STACKTRACES_STATIC_SYMBOLS: BOOL

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

HPX_WITH_THREAD_BACKTRACE_DEPTH: STRING

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

HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION: BOOL

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

HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES: BOOL

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

HPX_WITH_THREAD_CUMULATIVE_COUNTS: BOOL

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

HPX_WITH_THREAD_IDLE_RATES: BOOL

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

HPX_WITH_THREAD_LOCAL_STORAGE: BOOL

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

HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF: BOOL

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

HPX_WITH_THREAD_QUEUE_WAITTIME: BOOL

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

HPX_WITH_THREAD_STACK_MMAP: BOOL

Use mmap for stack allocation on appropriate platforms

HPX_WITH_THREAD_STEALING_COUNTS: BOOL

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

HPX_WITH_THREAD_TARGET_ADDRESS: BOOL

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

HPX_WITH_TIMER_POOL: BOOL

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

AGAS options

- **HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES: BOOL**

HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES: BOOL

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

Parcelport options

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

HPX_WITH_NETWORKING:BOOL

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

HPX_WITH_PARCELPORT_ACTION_COUNTERS:BOOL

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

HPX_WITH_PARCELPORT_LIBFABRIC:BOOL

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

HPX_WITH_PARCELPORT_MPI:BOOL

Enable the MPI based parcelport.

HPX_WITH_PARCELPORT_TCP:BOOL

Enable the TCP based parcelport.

HPX_WITH_PARCEL_PROFILING:BOOL

Enable profiling data for parcels

Profiling options

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

HPX_WITH_APEX:BOOL

Enable APEX instrumentation support.

HPX_WITH_ITTNOTIFY:BOOL

Enable Amplifier (ITT) instrumentation support.

HPX_WITH_PAPI:BOOL

Enable the PAPI based performance counter.

Debugging options

- *HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL*
- *HPX_WITH_PARALLEL_TESTS_BIND_NONE:BOOL*
- *HPX_WITH_SANITIZERS:BOOL*
- *HPX_WITH_TESTS_DEBUG_LOG:BOOL*
- *HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING*
- *HPX_WITH_TESTS_MAX_THREADS_PER_LOCALITY:STRING*

- *HPX_WITH_THREAD_DEBUG_INFO:BOOL*
- *HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL*
- *HPX_WITH_THREAD_GUARD_PAGE:BOOL*
- *HPX_WITH_VALGRIND:BOOL*
- *HPX_WITH_VERIFY_LOCKS:BOOL*
- *HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL*

HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL

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

HPX_WITH_PARALLEL_TESTS_BIND_NONE:BOOL

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

HPX_WITH_SANITIZERS:BOOL

Configure with sanitizer instrumentation support.

HPX_WITH_TESTS_DEBUG_LOG:BOOL

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

HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING

Destination for test debug logs (default: cout)

HPX_WITH_TESTS_MAX_THREADS_PER_LOCALITY:STRING

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

HPX_WITH_THREAD_DEBUG_INFO:BOOL

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

HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL

Use function address for thread description (default: OFF)

HPX_WITH_THREAD_GUARD_PAGE:BOOL

Enable thread guard page (default: ON)

HPX_WITH_VALGRIND:BOOL

Enable Valgrind instrumentation support.

HPX_WITH_VERIFY_LOCKS:BOOL

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

HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL

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

Modules options

- *HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL*
- *HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY:BOOL*
- *HPX_ITERATOR_SUPPORT_WITH_BOOST_ITERATOR_TRAVERSAL_TAG_COMPATIBILITY:BOOL*
- *HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS:BOOL*
- *HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION:BOOL*
- *HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE:BOOL*
- *HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL*

- **HPX_TOPOLOGY_WITH_ADDITIONAL_HWLOC_TESTING:BOOL**

HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL

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

HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY:BOOL

Enable Boost.FileSystem compatibility. (default: OFF)

HPX_ITERATOR_SUPPORT_WITH_BOOST_ITERATOR_TRAVERSAL_TAG_COMPATIBILITY:BOOL

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

HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS:BOOL

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

HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION:BOOL

Enable serializing raw pointers. (default: OFF)

HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE:BOOL

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

HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL

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

HPX_TOPOLOGY_WITH_ADDITIONAL_HWLOC_TESTING:BOOL

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

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

Additional tools and libraries used by HPX

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

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

BOOST_ROOT:PATH

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

HWLOC_ROOT:PATH

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

PAPI_ROOT:PATH

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

AMPLIFIER_ROOT:PATH

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

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

HDF5_ROOT:PATH

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

2.3.2 Creating HPX projects

Using HPX with pkg-config

How to build HPX applications with pkg-config

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

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

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

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

Now, in the directory where you put hello_world.cpp, issue the following commands (where \$HPX_LOCATION is the build directory or CMAKE_INSTALL_PREFIX you used while building *HPX*):

```
$ export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig
$ c++ -o hello_world hello_world.cpp \
`pkg-config --cflags --libs hpx_application` \
-lhpx_iostreams -DHPX_APPLICATION_NAME=hello_world
```

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

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

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

To test the program, type:

```
$ ./hello_world
```

which should print Hello World! and exit.

How to build HPX components with pkg-config

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

hello_world_component.cpp

```
#include <hpx/config.hpp>
#ifndef HPX_COMPUTE_DEVICE_CODE
#include "hello_world_component.hpp"
#include <hpx/iostream.hpp>

#include <iostream>

namespace examples { namespace server
{
    void hello_world::invoke()
    {
        hpx::cout << "Hello HPX World!" << std::endl;
    }
} }

Hpx_REGISTER_COMPONENT_MODULE()

typedef hpx::components::component<
    examples::server::hello_world
> hello_world_type;

Hpx_REGISTER_COMPONENT(hello_world_type, hello_world)

Hpx_REGISTER_ACTION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action)
#endif
```

hello_world_component.hpp

```
#pragma once

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

#include <utility>

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

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

#endif

```

hello_world_client.cpp

```

#include <hpx/config.hpp>
#if defined(HPX_COMPUTE_HOST_CODE)
#include <hpx/wrap_main.hpp>

#include "hello_world_component.hpp"

int main()
{
{
    // Create a single instance of the component on this locality.
    examples::hello_world client =
        hpx::new<examples::hello_world>(hpx::find_here());

    // Invoke the component's action, which will print "Hello World!".
    client.invoke();
}

return 0;
}
#endif

```

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

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

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

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

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

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

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

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

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

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

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

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

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

Using *HPX* with CMake-based projects

In addition to the `pkg-config` support discussed on the previous pages, *HPX* comes with full CMake support. In order to integrate *HPX* into existing or new `CMakeLists.txt`, you can leverage the `find_package`³⁵ command integrated into CMake. Following, is a Hello World component example using CMake.

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

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

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

³⁵ https://www.cmake.org/cmake/help/latest/command/find_package.html

```
# Require a recent version of cmake
cmake_minimum_required(VERSION 3.18 FATAL_ERROR)

# This project is C++ based.
project(your_app CXX)

# Instruct cmake to find the HPX settings
find_package(HPX)
```

In order to have CMake find *HPX*, it needs to be told where to look for the `HPXConfig.cmake` file that is generated when *HPX* is built or installed. It is used by `find_package(HPX)` to set up all the necessary macros needed to use *HPX* in your project. The ways to achieve this are:

- Set the `HPX_DIR` CMake variable to point to the directory containing the `HPXConfig.cmake` script on the command line when you invoke CMake:

```
$ cmake -DHPX_DIR=$HPX_LOCATION/lib/cmake/HPX ...
```

where `$HPX_LOCATION` is the build directory or `CMAKE_INSTALL_PREFIX` you used when building/configuring *HPX*.

- Set the `CMAKE_PREFIX_PATH` variable to the root directory of your *HPX* build or install location on the command line when you invoke CMake:

```
$ cmake -DCMAKE_PREFIX_PATH=$HPX_LOCATION ...
```

The difference between `CMAKE_PREFIX_PATH` and `HPX_DIR` is that CMake will add common postfixes, such as `lib/cmake/<project>`, to the `CMAKE_PREFIX_PATH` and search in these locations too. Note that if your project uses *HPX* as well as other CMake-managed projects, the paths to the locations of these multiple projects may be concatenated in the `CMAKE_PREFIX_PATH`.

- The variables above may be set in the CMake GUI or curses `ccmake` interface instead of the command line.

Additionally, if you wish to require *HPX* for your project, replace the `find_package(HPX)` line with `find_package(HPX REQUIRED)`.

You can check if *HPX* was successfully found with the `HPX_FOUND` CMake variable.

Using CMake targets

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

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

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

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

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

Creating a component requires setting two additional compile definitions:

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

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

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

Using macros to create new targets

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

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

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

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

The available options to `add_hpx_component` are:

- `SOURCES`: The source files for that component
- `HEADERS`: The header files for that component
- `DEPENDENCIES`: Other libraries or targets this component depends on
- `COMPONENT_DEPENDENCIES`: The components this component depends on
- `PLUGIN`: Treats this component as a plugin-able library
- `COMPILE_FLAGS`: Additional compiler flags
- `LINK_FLAGS`: Additional linker flags
- `FOLDER`: Adds the headers and source files to this Source Group folder

- EXCLUDE_FROM_ALL: Do not build this component as part of the all target

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

```
# build your application using HPX
add_hpx_executable(hello_world
    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_libraries³⁶ in your targets. The *HPX* include directories are available with the HPX_INCLUDE_DIRS CMake variable.

Using the *HPX* compiler wrapper hpxcxx

The hpxcxx compiler wrapper helps to compile a *HPX* component, application, or object file, based on the arguments passed to it.

```
$ hpxcxx [--exe=<APPLICATION_NAME> | --comp=<COMPONENT_NAME> | -c] FLAGS FILES
```

The hpxcxx command **requires** that either an application or a component is built or -c flag is specified. If the build is against a debug build, the -g is to be specified while building.

Optional FLAGS

- -l <LIBRARY> | -l<LIBRARY>: Links <LIBRARY> to the build
- -g: Specifies that the application or component build is against a debug build
- -rd: Sets release-with-debug-info option
- -mr: Sets minsize-release option

All other flags (like -o OUTPUT_FILE) are directly passed to the underlying C++ compiler.

³⁶ https://www.cmake.org/cmake/help/latest/command/target_link_libraries.html

Using macros to set up existing targets to use HPX

In addition to the `add_hpx_component` and `add_hpx_executable`, you can use the `hpx_setup_target` macro to have an already existing target to be used with the *HPX* libraries:

```
hpx_setup_target(target)
```

Optional parameters are:

- `EXPORT`: Adds it to the CMake export list `HPXTargets`
- `INSTALL`: Generates an install rule for the target
- `PLUGIN`: Treats this component as a plugin-able library
- `TYPE`: The type can be: `EXECUTABLE`, `LIBRARY` or `COMPONENT`
- `DEPENDENCIES`: Other libraries or targets this component depends on
- `COMPONENT_DEPENDENCIES`: The components this component depends on
- `COMPILE_FLAGS`: Additional compiler flags
- `LINK_FLAGS`: Additional linker flags

If you do not use CMake, you can still build against *HPX*, but you should refer to the section on *How to build HPX components with pkg-config*.

Note: Since *HPX* relies on dynamic libraries, the dynamic linker needs to know where to look for them. If *HPX* isn't installed into a path that is configured as a linker search path, external projects need to either set `RPATH` or adapt `LD_LIBRARY_PATH` to point to where the *HPX* libraries reside. In order to set `RPATH`s, you can include `HPX_SetFullRPATH` in your project after all libraries you want to link against have been added. Please also consult the CMake documentation [here³⁷](#).

Using HPX with Makefile

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

How to build HPX applications with makefile

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

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

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

³⁷ <https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/RPATH-handling>

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

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

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

CXXFLAGS=-O3 -std=c++17

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

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

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

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

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

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

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

To build the program, type:

```
$ make
```

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

```
$ ./hello_world
```

How to build HPX components with makefile

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

hello_world_component.cpp

```
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include "hello_world_component.hpp"
#include <hpx/iostream.hpp>

#include <iostream>
```

(continues on next page)

(continued from previous page)

```

namespace examples { namespace server
{
    void hello_world::invoke()
    {
        hpx::cout << "Hello HPX World!" << std::endl;
    }
}

HPX_REGISTER_COMPONENT_MODULE()

typedef hpx::components::component<
    examples::server::hello_world
> hello_world_type;

HPX_REGISTER_COMPONENT(hello_world_type, hello_world)

HPX_REGISTER_ACTION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action)
#endif

```

hello_world_component.hpp

```

#pragma once

#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/hpx.hpp>
#include <hpx/include/actions.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/include/components.hpp>
#include <hpx/serialization.hpp>

#include <utility>

namespace examples { namespace server
{
    struct HPX_COMPONENT_EXPORT hello_world
        : hpx::components::component_base<hello_world>
    {
        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)

```

(continues on next page)

(continued from previous page)

```

        : base_type(std::move(f))
    {}

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

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

#endif

```

hello_world_client.cpp

```

#include <hpx/config.hpp>
#if defined(HPX_COMPUTE_HOST_CODE)
#include <hpx/wrap_main.hpp>

#include "hello_world_component.hpp"

int main()
{
{
    // Create a single instance of the component on this locality.
    examples::hello_world client =
        hpx::new_<examples::hello_world>(hpx::find_here());

    // Invoke the component's action, which will print "Hello World!".
    client.invoke();
}

    return 0;
}
#endif

```

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

```

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

CXXFLAGS=-O3 -std=c++17

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

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

LIBRARY_DIRECTIVES=-L$ (HPX_ROOT)/lib $ (HPX_ROOT)/lib/libhpx_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

```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

```

To build the program, type:

```
$ make
```

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

```
$ ./hello_world
```

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

2.3.3 Starting the *HPX* runtime

In order to write an application which uses services from the *HPX* runtime system you need to initialize the *HPX* library by inserting certain calls into the code of your application. Depending on your use case, this can be done in 3 different ways:

- *Minimally invasive*: Re-use the `main()` function as the main *HPX* entry point.
- *Balanced use case*: Supply your own main *HPX* entry point while blocking the main thread.
- *Most flexibility*: Supply your own main *HPX* entry point while avoiding to block the main thread.
- *Suspend and resume*: As above but suspend and resume the *HPX* runtime to allow for other runtimes to be used.

Re-use the `main()` function as the main HPX entry point

This method is the least intrusive to your code. It however provides you with the smallest flexibility in terms of initializing the *HPX* runtime system. The following code snippet shows what a minimal *HPX* application using this technique looks like:

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

int main(int argc, char* argv[])
{
    return 0;
}
```

The only change to your code you have to make is to include the file `hpx/hpx_main.hpp`. In this case the function `main()` will be invoked as the first *HPX* thread of the application. The runtime system will be initialized behind the scenes before the function `main()` is executed and will automatically stop after `main()` has returned. For this method to work you must link your application to the CMake target `HPX::wrap_main`. This is done automatically if you are using the provided macros (*Using macros to create new targets*) to set up your application, but must be done explicitly if you are using targets directly (*Using CMake targets*). All *HPX* API functions can be used from within the `main()` function now.

Note: The function `main()` does not need to expect receiving `argc` and `argv` as shown above, but could expose the signature `int main()`. This is consistent with the usually allowed prototypes for the function `main()` in C++ applications.

All command line arguments specific to *HPX* will still be processed by the *HPX* runtime system as usual. However, those command line options will be removed from the list of values passed to `argc/argv` of the function `main()`. The list of values passed to `main()` will hold only the commandline options which are not recognized by the *HPX* runtime system (see the section *HPX Command Line Options* for more details on what options are recognized by *HPX*).

Note: In this mode all one-letter-shortcuts are disabled which are normally available on the *HPX* command line (such as `-t` or `-l` see *HPX Command Line Options*). This is done to minimize any possible interaction between the command line options recognized by the *HPX* runtime system and any command line options defined by the application.

The value returned from the function `main()` as shown above will be returned to the operating system as usual.

Important: To achieve this seamless integration, the header file `hpx/hpx_main.hpp` defines a macro:

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

which could result in unexpected behavior.

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

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

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

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

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

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

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

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

This method of invoking *HPX* has the advantage of you being able to decide which version of `hpx::init` to call. This allows to pass additional configuration parameters while initializing the *HPX* runtime system.

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

int hpx_main(int argc, char* argv[])
{
    // Any HPX application logic goes here...
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    // Initialize HPX, run hpx_main as the first HPX thread, and
    // wait for hpx::finalize being called.
    return hpx::init(argc, argv);
}
```

Note: The function `hpx_main` does not need to expect receiving `argc/argv` as shown above, but could expose one of the following signatures:

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

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

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

There are many additional overloads of `hpx::init` available, such as for instance to provide your own entry point function instead of `hpx_main`. Please refer to the function documentation for more details (see: `hpx/hpx_init.hpp`).

Supply your own main *HPX* entry point while avoiding to block the main thread

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

The thread executing the function `hpx::start` will *not* block waiting for the runtime system to exit, but will return immediately. The function `hpx::finalize` has to be called on one of the *HPX* localities in order to signal that all work has been scheduled and the runtime system should be stopped after the scheduled work has been executed.

This method of invoking *HPX* is useful for applications where the main thread is used for special operations, such 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>

int hpx_main(int argc, char* argv[])
{
    // Any HPX application logic goes here...
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    // Initialize HPX, run hpx_main.
    hpx::start(argc, argv);

    // ...Execute other code here...

    // Wait for hpx::finalize being called.
    return hpx::stop();
}
```

Note: The function `hpx_main` does not need to expect receiving `argc/argv` as shown above, but could expose one of the following signatures:

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

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

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

There are many additional overloads of `hpx::start` available, such as for instance to provide your own entry point function instead of `hpx_main`. Please refer to the function documentation for more details (see: `hpx/hpx_start.hpp`).

Suspending and resuming the *HPX* runtime

In some applications it is required to combine *HPX* with other runtimes. To support this use case *HPX* provides two functions: `hpx::suspend` and `hpx::resume`. `hpx::suspend` is a blocking call which will wait for all scheduled tasks to finish executing and then put the thread pool OS threads to sleep. `hpx::resume` simply wakes up the sleeping threads so that they are ready to accept new work. `hpx::suspend` and `hpx::resume` can be found in the header `hpx/hpx_suspend.hpp`.

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

int main(int argc, char* argv[])
{
    // Initialize HPX, don't run hpx_main
    hpx::start(nullptr, argc, argv);

    // Schedule a function on the HPX runtime
    hpx::apply(&my_function, ...);

    // Wait for all tasks to finish, and suspend the HPX runtime
    hpx::suspend();

    // Execute non-HPX code here

    // Resume the HPX runtime
    hpx::resume();

    // Schedule more work on the HPX runtime

    // hpx::finalize has to be called from the HPX runtime before hpx::stop
    hpx::apply([]() { hpx::finalize(); });
    return hpx::stop();
}
```

Note: `hpx::suspend` does not wait for `hpx::finalize` to be called. Only call `hpx::finalize` when you wish to fully stop the *HPX* runtime.

Warning:

`hpx::suspend` only waits for local tasks, i.e. tasks on the current locality, to finish executing. When using `hpx::suspend` in a multi-locality scenario the user is responsible for ensuring that any work required from other localities has also finished.

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

Automatically suspending worker threads

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

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

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

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

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

int main(int argc, char* argv[])
{
    // Initialize HPX, don't run hpx_main
    hpx::start(nullptr, argc, argv);

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

    // hpx::finalize has to be called from the HPX runtime before hpx::stop
    hpx::apply([]() { hpx::finalize(); });
    return hpx::stop();
}
```

In this example each call to `hpx::run_as_hpx_thread` acts as a “parallel region”.

Working of `hpx_main.hpp`

In order to initialize *HPX* from `main()`, we make use of linker tricks.

It is implemented differently for different Operating Systems. Method of implementation is as follows:

- *Linux*: Using linker `--wrap` option.
- *Mac OSX*: Using the linker `-e` option.
- *Windows*: Using `#define main hpx_startup::user_main`

Linux implementation

We make use of the Linux linker `ld`'s `--wrap` option to wrap the `main()` function. This way any call to `main()` are redirected to our own implementation of `main`. It is here that we check for the existence of `hpx_main.hpp` by making use of a shadow variable `include_libhpx_wrap`. The value of this variable determines the function stack at runtime.

The implementation can be found in `libhpx_wrap.a`.

Important: It is necessary that `hpx_main.hpp` be not included more than once. Multiple inclusions can result in multiple definition of `include_libhpx_wrap`.

Mac OSX implementation

Here we make use of yet another linker option `-e` to change the entry point to our custom entry function `initialize_main`. We initialize the *HPX* runtime system from this function and call `main` from the initialized system. We determine the function stack at runtime by making use of the shadow variable `include_libhpx_wrap`.

The implementation can be found in `libhpx_wrap.a`.

Important: It is necessary that `hpx_main.hpp` be not included more than once. Multiple inclusions can result in multiple definition of `include_libhpx_wrap`.

Windows implementation

We make use of a macro `#define main hpx_startup::user_main` to take care of the initializations.

This implementation could result in unexpected behaviors.

2.3.4 Launching and configuring *HPX* applications

Configuring *HPX* applications

All *HPX* applications can be configured using special command line options and/or using special configuration files. This section describes the available options, the configuration file format, and the algorithm used to locate possible predefined configuration files. Additionally this section describes the defaults assumed if no external configuration information is supplied.

During startup any *HPX* application applies a predefined search pattern to locate one or more configuration files. All found files will be read and merged in the sequence they are found into one single internal database holding all configuration properties. This database is used during the execution of the application to configure different aspects of the runtime system.

In addition to the ini files, any application can supply its own configuration files, which will be merged with the configuration database as well. Moreover, the user can specify additional configuration parameters on the command line when executing an application. The *HPX* runtime system will merge all command line configuration options (see the description of the `--hpx:ini`, `--hpx:config`, and `--hpx:app-config` command line options).

The HPX INI File Format

All *HPX* applications can be configured using a special file format which is similar to the well-known [Windows INI file format](#)³⁸. This is a structured text format allowing to group key/value pairs (properties) into sections. The basic element contained in an ini file is the property. Every property has a name and a value, delimited by an equals sign '='. The name appears to the left of the equals sign:

```
name=value
```

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

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

```
[section]
```

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

```
[outer_section.inner_section]
```

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

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

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

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

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

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

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

Note: Any property `${[section.key]:default}` is evaluated whenever it is queried and not when the configuration data is initialized. This allows for lazy evaluation and relaxes initialization order of different sections. The only exception are recursive property values, e.g. values referring to the very key they are associated with. Those property values are evaluated at initialization time to avoid infinite recursion.

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

Built-in Default Configuration Settings

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

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

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

The system configuration section

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

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

The hpx configuration section

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

[hpx.stacks]

(continues on next page)

(continued from previous page)

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


Property	Description
hpx.location	This is initialized to the id of the <i>locality</i> this application instance is running on.
hpx.component	Duplicates are discarded. This property can refer to a list of directories separated by ':' (Linux, Android, and MacOS) or using ';' (Windows).
hpx.master_ini	This is initialized to the list of default paths of the main hpx.ini configuration files. This property can refer to a list of directories separated by ':' (Linux, Android, and MacOS) or using ';' (Windows).
hpx.ini_path	This is initialized to the default path where <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).
hpx.os_threads	This setting reflects the number of OS-threads used for running <i>HPX</i> -threads. Defaults to number of detected cores (not hyperthreads/PUs).
hpx.localities	This setting reflects the number of localities the application is running on. Defaults to 1.
hpx.program_name	This setting reflects the program name of the application instance. Initialized from the command line argv[0].
hpx.cmd_line	This setting reflects the actual command line used to launch this application instance.
hpx.lock_detection	This setting verifies that no locks are being held while a <i>HPX</i> thread is suspended. This setting is applicable only if HPX_WITH_VERIFY_LOCKS is set during configuration in CMake.
hpx.throw_on_error	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 HPX_WITH_VERIFY_LOCKS is set during configuration in CMake. This setting has no effect if hpx.lock_detection=0.
hpx.minimal_deadlock_detection	This setting enables support for minimal deadlock detection for <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 HPX_WITH_THREAD_DEADLOCK_DETECTION is set during configuration in CMake.
hpx.spinlock_detection_limit	This setting verifies that spinlocks don't spin longer than specified using the hpx.spinlock_detection_limit. This setting is applicable only if HPX_WITH_SPINLOCK_DEADLOCK_DETECTION is set during configuration in CMake. By default this is set to 1 (for Debug builds) or to 0 (for Release, RelWithDebInfo, RelMinSize builds).
hpx.spinlock_limit	This setting specifies the upper limit of allowed number of spins that spinlocks are allowed to perform. This setting is applicable only if HPX_WITH_SPINLOCK_DEADLOCK_DETECTION is set during configuration in CMake. By default this is set to 1000000.
hpx.max_background_threads	This setting defines the number of threads in the scheduler which are used to execute background work. By default this is the same as the number of cores used for the scheduler.
hpx.max_idle_loops	By default this is defined by the preprocessor constant HPX_IDLE_LOOP_COUNT_MAX. This is an internal setting which you should change only if you know exactly what you are doing.
hpx.max_busy_loops	This setting defines the maximum value of the busy-loop counter in the scheduler. By default this is defined by the preprocessor constant HPX_BUSY_LOOP_COUNT_MAX. This is an internal setting which you should change only if you know exactly what you are doing.
hpx.max_idle_time	This setting defines the maximum time (in milliseconds) for the scheduler to sleep after being idle for hpx.max_idle_loop_count iterations. This setting is applicable only if HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF is set during configuration in CMake. By default this is defined by the preprocessor constant HPX_IDLE_BACKOFF_TIME_MAX. This is an internal setting which you should change only if you know exactly what you are doing.
hpx.exception_level	This setting defines the verbosity of exceptions. Valid values are integers. A setting of 2 or higher prints all available information. A setting of 1 leaves out the build configuration and environment variables. A setting of 0 or lower prints only the description of the thrown exception and the file name, function, and line number where the exception was thrown. The default value is 2 or the value of the environment variable HPX_EXCEPTION_VERBOSITY.
hpx.stack_size	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 HPX_SMALL_STACK_SIZE (defaults to 0x8000). This value is used for all <i>HPX</i> threads by default, except for the thread running hpx_main (which runs on a large stack).
hpx.stack_size_medium	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 HPX_MEDIUM_STACK_SIZE (defaults to 0x20000).

The `hpx.threadpools` configuration section

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

Property	Description
<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:0}
min_add_new_count = ${HPX_THREAD_QUEUE_MIN_ADD_NEW_COUNT:10}
max_add_new_count = ${HPX_THREAD_QUEUE_MAX_ADD_NEW_COUNT:10}
max_delete_count = ${HPX_THREAD_QUEUE_MAX_DELETE_COUNT:1000}
```

Property	Description
<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 always.
<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 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 AGAS 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>.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 component factory.
<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 component factory.
<code>hpx.components.<component_instance_name>.enabled</code>	This setting explicitly enables or disables the component. This is an optional property, <code>HPX</code> assumed that the component is enabled if it is not defined.

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

The `hpx.parcel` configuration section

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

(continues on next page)

(continued from previous page)

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

Property	Description
hpx.parcel.address	This property defines the default IP address to be used for the <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</code> ("127.0.0.1").
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</code> (7910).
hpx.parcel.bootstrap	This property defines which parcelport type should be used during application bootstrap. The default depends on the compile time preprocessor constant <code>HPX_PARCEL_BOOTSTRAP</code> ("tcp").
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</code> (512).
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</code> (4).
hpx.parcel.max_message_size	This property defines the maximum allowed message size which will be transferable through the <i>parcel</i> layer. The default depends on the compile time preprocessor constant <code>HPX_PARCEL_MAX_MESSAGE_SIZE</code> (100000000 bytes).
hpx.parcel.max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size which will be transferable through the <i>parcel</i> layer. The default depends on the compile time preprocessor constant <code>HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE</code> (1000000 bytes).
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 = ${HPX_HAVE_PARCELPORT_TCP:$[hpx.parcel.enabled]}
array_optimization = ${HPX_PARCEL_TCP_ARRAY_OPTIMIZATION:$[hpx.parcel.array_
˓optimization]}
zero_copy_optimization = ${HPX_PARCEL_TCP_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.zero_
˓copy_optimization]}
async_serialization = ${HPX_PARCEL_TCP_ASYNC_SERIALIZATION:$[hpx.parcel.async_
˓serialization]}
parcel_pool_size = ${HPX_PARCEL_TCP_PARCEL_POOL_SIZE:$[hpx.threadpools.parcel_pool_
˓size]}
```

(continues on next page)

(continued from previous page)

```
max_connections = ${HPX_PARCEL_TCP_MAX_CONNECTIONS:[hpx.parcel.max_connections]}
max_connections_per_locality = ${HPX_PARCEL_TCP_MAX_CONNECTIONS_PER_LOCALITY:[hpx.
    ↪parcel.max_connections_per_locality]}
max_message_size = ${HPX_PARCEL_TCP_MAX_MESSAGE_SIZE:[hpx.parcel.max_message_size]}
max_outbound_message_size = ${HPX_PARCEL_TCP_MAX_OUTBOUND_MESSAGE_SIZE:[hpx.parcel.
    ↪max_outbound_message_size]}
```

Property	Description
hpx.parcel. tcp.enable	Enable the use of the default TCP parcelport. Note that the initial bootstrap of the overall HPX application will be performed using the default TCP connections. This parcelport is enabled by default. This will be disabled only if MPI is enabled (see below).
hpx. parcel.tcp. array_optimization	This property defines whether this <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 hpx.parcel.array_optimization.
hpx. parcel.tcp. zero_copy_optimization	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 hpx.parcel.zero_copy_optimization.
hpx. parcel.tcp. async_serialization	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 hpx.parcel.async_serialization.
hpx. parcel.tcp. parcel_pool_size	The value of this property defines the number of OS-threads created for the internal parcel thread pool of the TCP <i>parcel</i> port. The default is taken from hpx.threadpools.parcel_pool_size.
hpx. parcel.tcp. max_connections	This property defines how many network connections between different localities are overall kept alive by each of <i>locality</i> . The default is taken from hpx.parcel.max_connections.
hpx. parcel.tcp. max_connections_per_locality	This property defines the maximum number of network connections that one <i>locality</i> will open to another <i>locality</i> . The default is taken from hpx.parcel.max_connections_per_locality.
hpx. parcel.tcp. max_message_size	This property defines the maximum allowed message size which will be transferable through the <i>parcel</i> layer. The default is taken from hpx.parcel.max_message_size.
hpx. parcel.tcp. max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size which will be transferable through the <i>parcel</i> layer. The default is taken from hpx.parcel.max_outbound_message_size.

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

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

(continues on next page)

(continued from previous page)

```
async_serialization = ${HPX_HAVE_PARCEL_MPI_ASYNC_SERIALIZATION:[hpx.parcel.async_
˓→serialization]}
parcel_pool_size = ${HPX_HAVE_PARCEL_MPI_PARCEL_POOL_SIZE:[hpx.threadpools.parcel_
˓→pool_size]}
max_connections = ${HPX_HAVE_PARCEL_MPI_MAX_CONNECTIONS:[hpx.parcel.max_
˓→connections]}
max_connections_per_locality = ${HPX_HAVE_PARCEL_MPI_MAX_CONNECTIONS_PER_LOCALITY:
˓→${[hpx.parcel.max_connections_per_locality]}}
max_message_size = ${HPX_HAVE_PARCEL_MPI_MAX_MESSAGE_SIZE:[hpx.parcel.max_message_
˓→size]}
max_outbound_message_size = ${HPX_HAVE_PARCEL_MPI_MAX_OUTBOUND_MESSAGE_SIZE:[hpx.
˓→parcel.max_outbound_message_size]}
```

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

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

The `hpx.commandline` configuration section

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

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

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


Property	Description
<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>
    {
        void print_greeting ()
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }

        // Component actions need to be declared, this also defines the
        // type 'print_greeting_action' representing the action.
        HPX_DEFINE_COMPONENT_ACTION(server, print_greeting, print_greeting_action);
    };
}

// Declare boilerplate code required for each of the component actions.
HPX_REGISTER_ACTION_DECLARATION(app::server::print_greeting_action);
```

The corresponding source file contains mainly macro invocations which define boilerplate code needed for *HPX* to function properly:

```
// file: $APP_ROOT/app_server.cpp
#include "app_server.hpp"

// Define boilerplate required once per component module.
HPX_REGISTER_COMPONENT_MODULE();

// Define factory object associated with our component of type 'app::server'.
HPX_REGISTER_COMPONENT(app::server, app_server);

// Define boilerplate code required for each of the component actions. Use the
// same argument as used for HPX_REGISTER_ACTION_DECLARATION above.
HPX_REGISTER_ACTION(app::server::print_greeting_action);
```

The following gives an example of how the component can be used. We create one instance of the `app::server` component on the current *locality* and invoke the exposed action `print_greeting_action` using the global id of the newly created instance. Note, that no special code is required to delete the component instance after it is not needed anymore. It will be deleted automatically when its last reference goes out of scope, here at the closing brace of the block surrounding the code:

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

(continues on next page)

(continued from previous page)

```

int hpx_main()
{
{
    // Create an instance of the app_server component on the current locality.
    hpx::naming:id_type app_server_instance =
        hpx::create_component<app::server>(hpx::find_here());

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

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

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

```

In order to make sure that the application will be able to use the component `app::server`, special configuration information must be passed to *HPX*. The 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 2.5: 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.	<code>HPX_LOGLEVEL</code>
AGAS	AGAS	Logging output generated by the AGAS subsystem	<code>HPX_AGAS_LOGLEVEL</code>
Application	APP	Logging generated by applications.	<code>HPX_APP_LOGLEVEL</code>

By default, all logging output is redirected to the console instance of an application, where it is collected and written to a file, one file for each logging category.

Each logging category can be customized at two levels, the parameters for each are stored in the ini configuration sections `hpx.logging.CATEGORY` and `hpx.logging.console.CATEGORY` (where `CATEGORY` is the category shortcut as listed in the table above). The former influences logging at the source *locality* and the latter modifies the logging behaviour for each of the categories at the console instance of an application.

Levels

All *HPX* logging output has seven different logging levels. These levels can be set explicitly or through environmental variables in the main *HPX* ini file as shown below. The logging levels and their associated integral values are shown in the table below, ordered from most verbose to least verbose. By default, all *HPX* logs are set to 0, e.g. all logging output is disabled by default.

Table 2.6: Logging levels

Logging level	Integral value
<debug>	5
<info>	4
<warning>	3
<error>	2
<fatal>	1
No logging	0

Tip: The easiest way to enable logging output is to set the environment variable corresponding to the logging category to an integral value as described in the table above. For instance, setting `HPX_LOGLEVEL=5` will enable full logging output for the general category. Please note that the syntax and means of setting environment variables varies between operating systems.

Configuration

Logs will be saved to destinations as configured by the user. By default, logging output is saved on the console instance of an application to `hpx.<CATEGORY>.<PID>.lo` (where `CATEGORY` and `PID` are placeholders for the category shortcut and the OS process id). The output for the general logging category is saved to `hpx.<PID>.log`. The default settings for the general logging category are shown here (the syntax is described in the section *The HPX INI File Format*):

```
[hpx.logging]
level = ${HPX_LOGLEVEL:0}
destination = ${HPX_LOGDESTINATION:console}
format = ${HPX_LOGFORMAT:(T%locality%/%hpxthread%.%hpxphase%/%hpxcomponent%) P
    ↪%parentloc%/%hpxparent%.%hpxparentphase% %time%($hh:$mm.$ss.$milis) [%idx%] |\\n}
```

The logging level is taken from the environment variable `HPX_LOGLEVEL` and defaults to zero, e.g. no logging. The default logging destination is read from the environment variable `HPX_LOGDESTINATION`. On any of the localities it defaults to `console` which redirects all generated logging output to the console instance of an application. The following table lists the possible destinations for any logging output. It is possible to specify more than one destination separated by whitespace.

Table 2.7: Logging destinations

Logging destination	Description
file(<filename>)	Direct all output to a file with the given <filename>.
cout	Direct all output to the local standard output of the application instance on this <i>locality</i> .
cerr	Direct all output to the local standard error output of the application instance on this <i>locality</i> .
console	Direct all output to the console instance of the application. The console instance has its logging destinations configured separately.
android_log	Direct all output to the (Android) system log (available on Android systems only).

The logging format is read from the environment variable `HPX_LOGFORMAT` and it defaults to a complex format description. This format consists of several placeholder fields (for instance `%locality%` which will be replaced by concrete values when the logging output is generated. All other information is transferred verbatim to the output. The table below describes the available field placeholders. The separator character `|` separates the logging message prefix formatted as shown and the actual log message which will replace the separator.

Table 2.8: Available field placeholders

Name	Description
<code>locality</code>	The id of the <i>locality</i> on which the logging message was generated.
<code>hpxthread</code>	The id of the <i>HPX</i> -thread generating this logging output.
<code>hpxphase</code>	The phase ⁴⁰ of the <i>HPX</i> -thread generating this logging output.
<code>hpxcomponent</code>	The local virtual address of the component which the current <i>HPX</i> -thread is accessing.
<code>parentloc</code>	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.
<code>hpxparent</code>	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.
<code>hpxparentphase</code>	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.
<code>time</code>	The time stamp for this logging output line as generated by the source <i>locality</i> .
<code>idx</code>	The sequence number of the logging output line as generated on the source <i>locality</i> .
<code>osthread</code>	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 [00000000000004d]
    <info> [RT] successfully created component {000000100ff0001, 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.

⁴⁰ The phase of a *HPX*-thread counts how often this thread has been activated.

HPX Command Line Options

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

HPX options (allowed on command line only)

```
--hpx:help
    print out program usage (default: this message), possible values: full (additionally prints options from components)
--hpx:version
    print out HPX version and copyright information
--hpx:info
    print out HPX configuration information
--hpx:options-file arg
    specify a file containing command line options (alternatively: @filepath)
```

HPX options (additionally allowed in an options file)

```
--hpx:worker
    run this instance in worker mode
--hpx:console
    run this instance in console mode
--hpx:connect
    run this instance in worker mode, but connecting late
--hpx:run-agas-server
    run AGAS server as part of this runtime instance
--hpx:run-hpx-main
    run the hpx_main function, regardless of locality mode
--hpx:hpx arg
    the IP address the HPX parcelport is listening on, expected format: address:port (default: 127.0.0.1:7910)
--hpx:agas arg
    the IP address the AGAS root server is running on, expected format: address:port (default: 127.0.0.1:7910)
--hpx:run-agas-server-only
    run only the AGAS server
--hpx:nodofile arg
    the file name of a node file to use (list of nodes, one node name per line and core)
--hpx:nodes arg
    the (space separated) list of the nodes to use (usually this is extracted from a node file)
--hpx:endnodes
    this can be used to end the list of nodes specified using the option --hpx:nodes
--hpx:ifsuffix arg
    suffix to append to host names in order to resolve them to the proper network interconnect
```

--**hpx:ifprefix** arg
prefix to prepend to host names in order to resolve them to the proper network interconnect

--**hpx:iftransform** arg
sed-style search and replace (s/search/replace/) used to transform host names to the proper network interconnect

--**hpx:localities** arg
the number of localities to wait for at application startup (default: 1)

--**hpx:node** arg
number of the node this *locality* is run on (must be unique)

--**hpx:ignore-batch-env**
ignore batch environment variables

--**hpx:expect-connecting-localities**
this *locality* expects other localities to dynamically connect (this is implied if the number of initial localities is larger than 1)

--**hpx:pu-offset**
the first processing unit this instance of *HPX* should be run on (default: 0)

--**hpx:pu-step**
the step between used processing unit numbers for this instance of *HPX* (default: 1)

--**hpx:threads** arg
the number of operating system threads to spawn for this *HPX locality*. Possible values are: numeric values 1, 2, 3 and so on, *all* (which spawns one thread per processing unit, includes hyperthreads), or *cores* (which spawns one thread per core) (default: *cores*).

--**hpx:cores** arg
the number of cores to utilize for this *HPX locality* (default: *all*, i.e. the number of cores is based on the number of threads --*hpx:threads* assuming --*hpx:bind=compact*)

--**hpx:affinity** arg
the affinity domain the OS threads will be confined to, possible values: *pu*, *core*, *numa*, *machine* (default: *pu*)

--**hpx:bind** arg
the detailed affinity description for the OS threads, see *More details about HPX command line options* for a detailed description of possible values. Do not use with --*hpx:pu-step*, --*hpx:pu-offset* or --*hpx:affinity* options. Implies --*hpx:numa-sensitive* (--*hpx:bind=none*) disables defining thread affinities.

--**hpx:use-process-mask**
use the process mask to restrict available hardware resources (implies --*hpx:ignore-batch-env*)

--**hpx:print-bind**
print to the console the bit masks calculated from the arguments specified to all --*hpx:bind* options.

--**hpx:queuing** arg
the queue scheduling policy to use, options are *local*, *local-priority-fifo*, *local-priority-lifo*, *static*, *static-priority*, *abp-priority-fifo* and *abp-priority-lifo* (default: *local-priority-fifo*)

--**hpx:high-priority-threads** arg
the number of operating system threads maintaining a high priority queue (default: number of OS threads), valid for --*hpx:queuing=abp-priority*, --*hpx:queuing=static-priority* and --*hpx:queuing=local-priority* only

```
--hpx: numa-sensitive
    makes the scheduler NUMA sensitive
```

HPX configuration options

```
--hpx:app-config arg
    load the specified application configuration (ini) file
--hpx:config arg
    load the specified hpx configuration (ini) file
--hpx:ini arg
    add a configuration definition to the default runtime configuration
--hpx:exit
    exit after configuring the runtime
```

HPX debugging options

```
--hpx:list-symbolic-names
    list all registered symbolic names after startup
--hpx:list-component-types
    list all dynamic component types after startup
--hpx:dump-config-initial
    print the initial runtime configuration
--hpx:dump-config
    print the final runtime configuration
--hpx:debug-hpx-log [arg]
    enable all messages on the HPX log channel and send all HPX logs to the target destination (default: cout)
--hpx:debug-agas-log [arg]
    enable all messages on the AGAS log channel and send all AGAS logs to the target destination (default: cout)
--hpx:debug-parcel-log [arg]
    enable all messages on the parcel transport log channel and send all parcel transport logs to the target destination
        (default: cout)
--hpx:debug-timing-log [arg]
    enable all messages on the timing log channel and send all timing logs to the target destination (default: cout)
--hpx:debug-app-log [arg]
    enable all messages on the application log channel and send all application logs to the target destination (default:
        cout)
--hpx:debug-clp
    debug command line processing
--hpx:attach-debugger arg
    wait for a debugger to be attached, possible arg values: startup or exception (default: startup)
```

HPX options related to performance counters

--hpx:print-counter

print the specified performance counter either repeatedly and/or at the times specified by
--hpx:print-counter-at (see also option --hpx:print-counter-interval)

--hpx:print-counter-reset

print the specified performance counter either repeatedly and/or at the times specified by
--hpx:print-counter-at reset the counter after the value is queried. (see also option
--hpx:print-counter-interval)

--hpx:print-counter-interval

print the performance counter(s) specified with --hpx:print-counter repeatedly after the time interval
(specified in milliseconds), (default: 0, which means print once at shutdown)

--hpx:print-counter-destination

print the performance counter(s) specified with --hpx:print-counter to the given file (default:
console)

--hpx:list-counters

list the names of all registered performance counters, possible values: minimal (prints counter name skeletons), full (prints all available counter names)

--hpx:list-counter-infos

list the description of all registered performance counters, possible values: minimal (prints info for counter name skeletons), full (prints all available counter infos)

--hpx:print-counter-format

print the performance counter(s) specified with --hpx:print-counter possible formats in csv format
with header or without any header (see option --hpx:no-csv-header, possible values: csv (prints
counter values in CSV format with full names as header), csv-short (prints counter values in CSV for-
mat with shortnames provided with --hpx:print-counter as --hpx:print-counter shortname,
full-countername

--hpx:no-csv-header

print the performance counter(s) specified with --hpx:print-counter and csv or csv-short format
specified with --hpx:print-counter-format without header

--hpx:print-counter-at arg

print the performance counter(s) specified with --hpx:print-counter (or
--hpx:print-counter-reset at the given point in time, possible argument values: startup,
shutdown (default), noshutdown

--hpx:reset-counters

reset all performance counter(s) specified with --hpx:print-counter after they have been evaluated.

--hpx:print-counters-locally

Each *locality* prints only its own local counters. If this is used with
--hpx:print-counter-destination=<file>, the code will append a ".<locality_id>"
to the file name in order to avoid clashes between localities.

Command line argument shortcuts

Additionally, the following shortcuts are available from every *HPX* application.

Table 2.9: Predefined command line option shortcuts

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

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

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

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

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

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

would be expanded to:

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

Important: Any shortcut option should either start with a single '-' or with two '--' characters. Shortcuts

starting with a single '-' are interpreted as short options (i.e. everything after the first character following the '-' is treated as the argument). Shortcuts starting with '--' are interpreted as long options. No other shortcut formats are supported.

Specifying options for single localities only

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

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

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

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

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

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

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

More details about *HPX* command line options

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

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

The command line option `--hpx:bind`

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

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

The specified affinities refer to specific regions within a machine hardware topology. In order to understand the hardware topology of a particular machine it may be useful to run the `lstopo` tool which is part of Portable Hardware

Locality (HWLOC) to see the reported topology tree. Seeing and understanding a topology tree will definitely help in understanding the concepts that are discussed below.

Affinities can be specified using HWLOC (Portable Hardware Locality (HWLOC)) tuples. Tuples of HWLOC *objects* and associated *indexes* can be specified in the form `object:index`, `object:index-index` or `object:index,...,index`. HWLOC objects represent types of mapped items in a topology tree. Possible values for objects are `socket`, `numanode`, `core` and `pu` (processing unit). Indexes are non-negative integers that specify a unique physical object in a topology tree using its logical sequence number.

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

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

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

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

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

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

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

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

Here `thread:0-3` specifies the OS threads for which to define affinity bindings, and `core:0-3.pu: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.

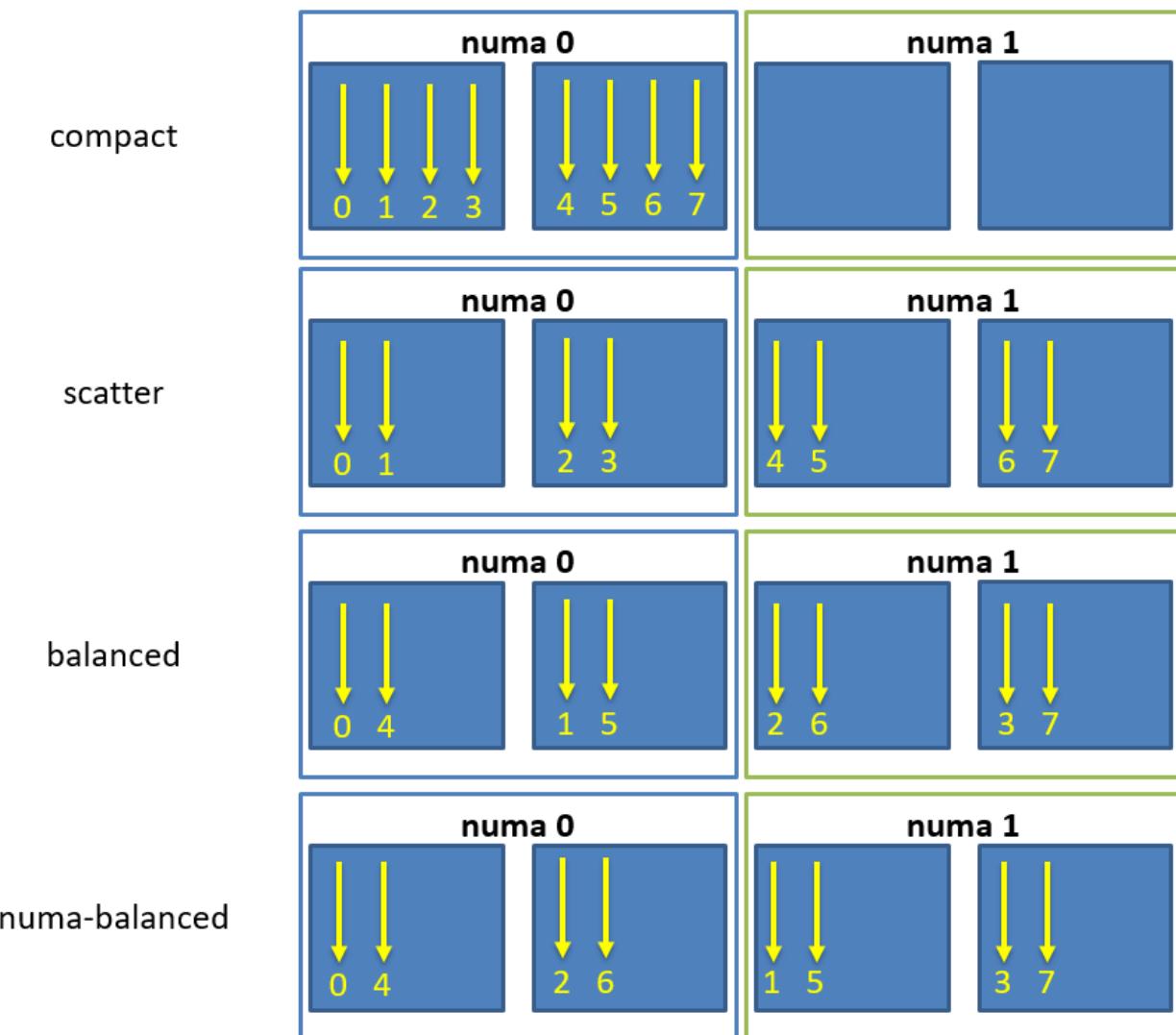


Fig. 2.7: Schematic of thread affinity type distributions.

In addition to the predefined distributions it is possible to restrict the resources used by *HPX* to the process CPU

mask. The CPU mask is typically set by e.g. MPI³⁹ and batch environments. Using the command line option `--hpx:use-process-mask` makes *HPX* act as if only the processing units in the CPU mask are available for use by *HPX*. The number of threads is automatically determined from the CPU mask. The number of threads can still be changed manually using this option, but only to a number less than or equal to the number of processing units in the CPU mask. The option `--hpx:print-bind` is useful in conjunction with `--hpx:use-process-mask` to make sure threads are placed as expected.

2.3.5 Writing single-node *HPX* applications

HPX is a C++ Standard Library for Concurrency and Parallelism. This means that it implements all of the corresponding facilities as defined by the C++ Standard. Additionally, *HPX* implements functionalities proposed as part of the ongoing C++ standardization process. This section focuses on the features available in *HPX* for parallel and concurrent computation on a single node, although many of the features presented here are also implemented to work in the distributed case.

Using LCOs

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

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

Channels

Channels combine communication (the exchange of a value) with synchronization (guaranteeing that two calculations (tasks) are in a known state). A channel can transport any number of values of a given type from a sender to a receiver:

```
hpx::lcos::local::channel<int> c;
hpx::future<int> f = c.get();
HPX_ASSERT(!f.is_ready());
c.set(42);
HPX_ASSERT(f.is_ready());
std::cout << f.get() << std::endl;
```

Channels can be handed to another thread (or in case of channel components, to other localities), thus establishing a communication channel between two independent places in the program:

```
void do_something(hpx::lcos::local::receive_channel<int> c,
                  hpx::lcos::local::send_channel<> done)
{
    // prints 43
    std::cout << c.get(hpx::launch::sync) << std::endl;
    // signal back
    done.set();
}
```

(continues on next page)

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

(continued from previous page)

```
void send_receive_channel()
{
    hpx::lcos::local::channel<int> c;
    hpx::lcos::local::channel<> done;

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

    // send some value
    c.set(43);
    // wait for thread to be done
    done.get().wait();
}
```

Note how `hpx::lcos::local::channel::get` without any arguments returns a future which is ready when a value has been set on the channel. The launch policy `hpx::launch::sync` can be used to make `hpx::lcos::local::channel::get` block until a value is set and return the value directly.

A channel component is created on one *locality* and can be sent to another *locality* using an action. This example also demonstrates how a channel can be used as a range of values:

```
// channel components need to be registered for each used type (not needed
// for hpx::lcos::local::channel)
HPX_REGISTER_CHANNEL(double)

void channel_sender(hpx::lcos::channel<double> c)
{
    for (double d : c)
        hpx::cout << d << std::endl;
}
HPX_PLAIN_ACTION(channel_sender)

void channel()
{
    // create the channel on this locality
    hpx::lcos::channel<double> c(hpx::find_here());

    // pass the channel to a (possibly remote invoked) action
    hpx::apply(channel_sender_action(), hpx::find_here(), c);

    // send some values to the receiver
    std::vector<double> v = {1.2, 3.4, 5.0};
    for (double d : v)
        c.set(d);

    // explicitly close the communication channel (implicit at destruction)
    c.close();
}
```

Composable guards

Composable guards operate in a manner similar to locks, but are applied only to asynchronous functions. The guard (or guards) is automatically locked at the beginning of a specified task and automatically unlocked at the end. Because guards are never added to an existing task's execution context, the calling of guards is freely composable and can never deadlock.

To call an application with a single guard, simply declare the guard and call run_guarded() with a function (task):

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

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

```
boost::shared<hpx::lcos::local::guard> gu1(new hpx::lcos::local::guard());
boost::shared<hpx::lcos::local::guard> gu2(new hpx::lcos::local::guard());
gs.add(*gu1);
gs.add(*gu2);
run_guarded(gs,task);
```

Guards use two atomic operations (which are not called repeatedly) to manage what they do, so overhead should be extremely low. The following guards are available in *HPX*:

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

Extended facilities for futures

Concurrency is about both decomposing and composing the program from the parts that work well individually and together. It is in the composition of connected and multicore components where today's C++ libraries are still lacking.

The functionality of `std::future` offers a partial solution. It allows for the separation of the initiation of an operation and the act of waiting for its result; however, the act of waiting is synchronous. In communication-intensive code this act of waiting can be unpredictable, inefficient and simply frustrating. The example below illustrates a possible synchronous wait using futures:

```
#include <future>
using namespace std;
int main()
{
    future<int> f = async([]() { return 123; });
    int result = f.get(); // might block
}
```

For this reason, *HPX* implements a set of extensions to `std::future` (as proposed by [__cpp11_n4107__](#)). This proposal introduces the following key asynchronous operations to `hpx::future`, `hpx::shared_future` and `hpx::async`, which enhance and enrich these facilities.

Table 2.11: Facilities extending `std::future`

Facility	Description
<code>hpx::future</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.
<code>unwrapping constructor for 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 users must handle exceptions and it may cause a blocking call. Unwrapping can allow users to mitigate this problem by doing an asynchronous call to unwrap the outermost future.
<code>hpx::fut</code>	These are soft cases There are soft cases 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 that holds a pre-computed result in its shared state. In the current standard it is non-trivial to create a future directly from a value. First a promise must be created, then the promise is set, and lastly the future is retrieved from the promise. This can now be done with one operation.

The standard also omits the ability to compose multiple futures. This is a common pattern that is ubiquitous in other asynchronous frameworks and is absolutely necessary in order to make C++ a powerful asynchronous programming language. Not including these functions is synonymous to Boolean algebra without AND/OR.

In addition to the extensions proposed by [N4313⁴¹](#), *HPX* adds functions allowing users to compose several futures in a more flexible way.

⁴¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

Table 2.12: 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.	N4313⁴² , ..._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.	N4313⁴³ , ..._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 nfinish.	<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 nfinish.	<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 nfinish 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, there are currently several proposals targeting different facilities supporting parallel programming. *HPX* implements (and extends) some of those proposals. This is well aligned with our strategy to align the APIs exposed from *HPX* with current and future C++ Standards.

At this point, *HPX* implements several of the C++ Standardization working papers, most notably [N4409⁴⁴](#) (Working Draft, Technical Specification for C++ Extensions for Parallelism), [N4411⁴⁵](#) (Task Blocks), and [N4406⁴⁶](#) (Parallel Algorithms Need Executors).

Using parallel algorithms

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

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

All parallel algorithms are very similar in semantics to their sequential counterparts (as defined in the namespace `std`) with an additional formal template parameter named `ExecutionPolicy`. The execution policy is generally

⁴² <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

⁴³ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

⁴⁴ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf>

⁴⁵ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

⁴⁶ <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::execution::sequenced_policy` or `hpx::execution::sequenced_task_policy` execute in sequential order. For `hpx::execution::sequenced_policy` the execution happens in the calling thread.

The applications of function objects in parallel algorithms invoked with an execution policy object of type `hpx::execution::parallel_policy` or `hpx::execution::parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and are indeterminately sequenced within each thread.

Important: It is the caller's responsibility to ensure correctness, such as making sure that the invocation does not introduce data races or deadlocks.

The applications of function objects in parallel algorithms invoked with an execution policy of type `hpx::execution::parallel_unsequenced_policy` is, in HPX, equivalent to the use of the execution policy `hpx::execution::parallel_policy`.

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

Parallel exceptions

During the execution of a standard parallel algorithm, if temporary memory resources are required by any of the algorithms and no memory is available, the algorithm throws a `std::bad_alloc` exception.

During the execution of any of the parallel algorithms, if the application of a function object terminates with an uncaught exception, the behavior of the program is determined by the type of execution policy used to invoke the algorithm:

- If the execution policy object is of type `hpx::execution::parallel_unsequenced_policy`, `hpx::terminate` shall be called.
- If the execution policy object is of type `hpx::execution::sequenced_policy`, `hpx::execution::sequenced_task_policy`, `hpx::execution::parallel_policy`, or `hpx::execution::parallel_task_policy`, the execution of the algorithm terminates with an `hpx::exception_list` exception. All uncaught exceptions thrown during the application of user-provided function objects shall be contained in the `hpx::exception_list`.

For example, the number of invocations of the user-provided function object in `for_each` is unspecified. When `hpx::parallel::v1::for_each` is executed sequentially, only one exception will be contained in the `hpx::exception_list` object.

These guarantees imply that, unless the algorithm has failed to allocate memory and terminated with `std::bad_alloc`, all exceptions thrown during the execution of the algorithm are communicated to the caller. It is unspecified whether an algorithm implementation will “forge ahead” after encountering and capturing a user exception.

The algorithm may terminate with the `std::bad_alloc` exception even if one or more user-provided function objects have terminated with an exception. For example, this can happen when an algorithm fails to allocate memory

while creating or adding elements to the `hpx::exception_list` object.

Parallel algorithms

HPX provides implementations of the following parallel algorithms:

Table 2.13: Non-modifying parallel algorithms (in header: <hpx/algorithm.hpp>)

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::adjacent_find</code>	Computes the differences between adjacent elements in a range.	<hpx/algorithm.hpp>	adjacent_find⁴⁷
<code>hpx::all_of</code>	Checks if a predicate is true for all of the elements in a range.	<hpx/algorithm.hpp>	all_any_none_of⁴⁸
<code>hpx::any_of</code>	Checks if a predicate is true for any of the elements in a range.	<hpx/algorithm.hpp>	all_any_none_of⁴⁹
<code>hpx::count</code>	Returns the number of elements equal to a given value.	<hpx/algorithm.hpp>	count⁵⁰
<code>hpx::count_if</code>	Returns the number of elements satisfying a specific criteria.	<hpx/algorithm.hpp>	count_if⁵¹
<code>hpx::equal</code>	Determines if two sets of elements are the same.	<hpx/algorithm.hpp>	equal⁵²
<code>hpx::find</code>	Finds the first element equal to a given value.	<hpx/algorithm.hpp>	find⁵³
<code>hpx::find_end</code>	Finds the last sequence of elements in a certain range.	<hpx/algorithm.hpp>	find_end⁵⁴
<code>hpx::find_first_of</code>	Searches for any one of a set of elements.	<hpx/algorithm.hpp>	find_first_of⁵⁵
<code>hpx::find_if</code>	Finds the first element satisfying a specific criteria.	<hpx/algorithm.hpp>	find_if⁵⁶
<code>hpx::find_if_not</code>	Finds the first element not satisfying a specific criteria.	<hpx/algorithm.hpp>	find_if_not⁵⁷
<code>hpx::for_each</code>	Applies a function to a range of elements.	<hpx/algorithm.hpp>	for_each⁵⁸
<code>hpx::for_each_n</code>	Applies a function to a number of elements.	<hpx/algorithm.hpp>	for_each_n⁵⁹
<code>hpx::lexicographical_compare</code>	Checks if a range of values is lexicographically less than another range of values.	<hpx/algorithm.hpp>	lexicographical_compare⁶⁰
<code>hpx::parallel::v1::mismatch</code>	Finds the first position where two ranges differ.	<hpx/algorithm.hpp>	mismatch⁶¹
<code>hpx::none_of</code>	Checks if a predicate is true for none of the elements in a range.	<hpx/algorithm.hpp>	all_any_none_of⁶²
<code>hpx::search</code>	Searches for a range of elements.	<hpx/algorithm.hpp>	search⁶³
2.3. Manual		hpp>	109
<code>hpx::search_n</code>	Searches for a number consecutive copies of an element in a range.	<hpx/algorithm.hpp>	search_n⁶⁴

⁴⁷ http://en.cppreference.com/w/cpp/algorithm/adjacent_find
⁴⁸ http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
⁴⁹ http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
⁵⁰ <http://en.cppreference.com/w/cpp/algorithm/count>
⁵¹ http://en.cppreference.com/w/cpp/algorithm/count_if
⁵² <http://en.cppreference.com/w/cpp/algorithm/equal>
⁵³ <http://en.cppreference.com/w/cpp/algorithm/find>
⁵⁴ http://en.cppreference.com/w/cpp/algorithm/find_end
⁵⁵ http://en.cppreference.com/w/cpp/algorithm/find_first_of
⁵⁶ http://en.cppreference.com/w/cpp/algorithm/find_if
⁵⁷ http://en.cppreference.com/w/cpp/algorithm/find_if_not
⁵⁸ http://en.cppreference.com/w/cpp/algorithm/for_each
⁵⁹ http://en.cppreference.com/w/cpp/algorithm/for_each_n
⁶⁰ http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
⁶¹ <http://en.cppreference.com/w/cpp/algorithm/mismatch>
⁶² http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
⁶³ <http://en.cppreference.com/w/cpp/algorithm/search>
⁶⁴ http://en.cppreference.com/w/cpp/algorithm/search_n

Table 2.14: Modifying parallel algorithms (In Header:
<hpx/algorithm.hpp>)

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::copy</code>	Copies a range of elements to a new location.	<code><hpx/algorithm.hpp></code>	<code>exclusive_scan</code> ⁶⁵
<code>hpx::copy_n</code>	Copies a number of elements to a new location.	<code><hpx/algorithm.hpp></code>	<code>copy_n</code> ⁶⁶
<code>hpx::copy_if</code>	Copies the elements from a range to a new location for which the given predicate is <code>true</code>	<code><hpx/algorithm.hpp></code>	<code>copy</code> ⁶⁷
<code>hpx::move</code>	Moves a range of elements to a new location.	<code><hpx/algorithm.hpp></code>	<code>move</code> ⁶⁸
<code>hpx::fill</code>	Assigns a range of elements a certain value.	<code><hpx/algorithm.hpp></code>	<code>fill</code> ⁶⁹
<code>hpx::fill_n</code>	Assigns a value to a number of elements.	<code><hpx/algorithm.hpp></code>	<code>fill_n</code> ⁷⁰
<code>hpx::generate</code>	Saves the result of a function in a range.	<code><hpx/algorithm.hpp></code>	<code>generate</code> ⁷¹
<code>hpx::generate_n</code>	Saves the result of N applications of a function.	<code><hpx/algorithm.hpp></code>	<code>generate_n</code> ⁷²
<code>hpx::remove</code>	Removes the elements from a range that are equal to the given value.	<code><hpx/algorithm.hpp></code>	<code>remove</code> ⁷³
<code>hpx::remove_if</code>	Removes the elements from a range that are equal to the given predicate is <code>false</code>	<code><hpx/algorithm.hpp></code>	<code>remove</code> ⁷⁴
<code>hpx::remove_copy</code>	Copies the elements from a range to a new location that are not equal to the given value.	<code><hpx/algorithm.hpp></code>	<code>remove_copy</code> ⁷⁵
<code>hpx::remove_copy_if</code>	Copies the elements from a range to a new location for which the given predicate is <code>false</code>	<code><hpx/algorithm.hpp></code>	<code>remove_copy</code> ⁷⁶
<code>hpx::replace</code>	Replaces all values satisfying specific criteria with another value.	<code><hpx/algorithm.hpp></code>	<code>replace</code> ⁷⁷
<code>hpx::replace_if</code>	Replaces all values satisfying specific criteria with another value.	<code><hpx/algorithm.hpp></code>	<code>replace</code> ⁷⁸
<code>hpx::replace_copy</code>	Copies a range, replacing elements satisfying specific criteria with another value.	<code><hpx/algorithm.hpp></code>	<code>replace_copy</code> ⁷⁹
<code>hpx::replace_copy_if</code>	Copies a range, replacing elements satisfying specific criteria with another value.	<code><hpx/algorithm.hpp></code>	<code>replace_copy</code> ⁸⁰
<code>hpx::reverse</code>	Reverses the order elements in a range.	<code><hpx/algorithm.hpp></code>	<code>reverse</code> ⁸¹
2.3. Manual		<code><hpx/algorithm.hpp></code>	111
<code>hpx::reverse_copy</code>	Creates a copy of a range that is reversed.	<code><hpx/algorithm.hpp></code>	<code>reverse_copy</code> ⁸²

Table 2.15: Set operations on sorted sequences (In Header:
<hpx/algorithm.hpp>)

Name	Description	In header	Algorithm page at cp-preference.com
<i>hpx::merge</i>	Merges two sorted ranges.	<i><hpx/algorithm.hpp></i>	merge⁹¹
<i>hpx::inplace_merge</i>	Merges two ordered ranges in-place.	<i><hpx/algorithm.hpp></i>	inplace_merge⁹²
<i>hpx::includes</i>	Returns true if one set is a subset of another.	<i><hpx/algorithm.hpp></i>	includes⁹³
<i>hpx::set_difference</i>	Computes the difference between two sets.	<i><hpx/algorithm.hpp></i>	set_difference⁹⁴
<i>hpx::set_intersection</i>	Computes the intersection of two sets.	<i><hpx/algorithm.hpp></i>	set_intersection⁹⁵
<i>hpx::set_symmetric_difference</i>	Computes the symmetric difference between two sets.	<i><hpx/algorithm.hpp></i>	set_symmetric_difference⁹⁶
<i>hpx::set_union</i>	Computes the union of two sets.	<i><hpx/algorithm.hpp></i>	set_union⁹⁷

⁶⁵ http://en.cppreference.com/w/cpp/algorithm/exclusive_scan

⁶⁶ http://en.cppreference.com/w/cpp/algorithm/copy_n

⁶⁷ <http://en.cppreference.com/w/cpp/algorithm/copy>

⁶⁸ <http://en.cppreference.com/w/cpp/algorithm/move>

⁶⁹ <http://en.cppreference.com/w/cpp/algorithm/fill>

⁷⁰ http://en.cppreference.com/w/cpp/algorithm/fill_n

⁷¹ <http://en.cppreference.com/w/cpp/algorithm/generate>

⁷² http://en.cppreference.com/w/cpp/algorithm/generate_n

⁷³ <http://en.cppreference.com/w/cpp/algorithm/remove>

⁷⁴ <http://en.cppreference.com/w/cpp/algorithm/remove>

⁷⁵ http://en.cppreference.com/w/cpp/algorithm/remove_copy

⁷⁶ http://en.cppreference.com/w/cpp/algorithm/remove_copy

⁷⁷ <http://en.cppreference.com/w/cpp/algorithm/replace>

⁷⁸ <http://en.cppreference.com/w/cpp/algorithm/replace>

⁷⁹ http://en.cppreference.com/w/cpp/algorithm/replace_copy

⁸⁰ http://en.cppreference.com/w/cpp/algorithm/replace_copy

⁸¹ <http://en.cppreference.com/w/cpp/algorithm/reverse>

⁸² http://en.cppreference.com/w/cpp/algorithm/reverse_copy

⁸³ <http://en.cppreference.com/w/cpp/algorithm/rotate>

⁸⁴ http://en.cppreference.com/w/cpp/algorithm/rotate_copy

⁸⁵ http://en.cppreference.com/w/cpp/algorithm/shift_left

⁸⁶ http://en.cppreference.com/w/cpp/algorithm/shift_right

⁸⁷ http://en.cppreference.com/w/cpp/algorithm/swap_ranges

⁸⁸ <http://en.cppreference.com/w/cpp/algorithm/transform>

⁸⁹ <http://en.cppreference.com/w/cpp/algorithm/unique>

⁹⁰ http://en.cppreference.com/w/cpp/algorithm/unique_copy

⁹¹ <http://en.cppreference.com/w/cpp/algorithm/merge>

⁹² http://en.cppreference.com/w/cpp/algorithm/inplace_merge

⁹³ <http://en.cppreference.com/w/cpp/algorithm/includes>

⁹⁴ http://en.cppreference.com/w/cpp/algorithm/set_difference

⁹⁵ http://en.cppreference.com/w/cpp/algorithm/set_intersection

⁹⁶ http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference

⁹⁷ http://en.cppreference.com/w/cpp/algorithm/set_union

Table 2.16: Heap operations (In Header: <hpx/algorithm.hpp>)

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::is_heap</code>	Returns <code>true</code> if the range is max heap.	<hpx/algorithm.hpp>	<code>is_heap</code> ⁹⁸
<code>hpx::is_heap_until</code>	Returns the first element that breaks a max heap.	<hpx/algorithm.hpp>	<code>is_heap_until</code> ⁹⁹
<code>hpx::make_heap</code>	Constructs a max heap in the range [first, last).	<hpx/algorithm.hpp>	<code>make_heap</code> ¹⁰⁰

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

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::max_element</code>	Returns the largest element in a range.	<hpx/algorithm.hpp>	<code>max_element</code> ¹⁰¹
<code>hpx::min_element</code>	Returns the smallest element in a range.	<hpx/algorithm.hpp>	<code>min_element</code> ¹⁰²
<code>hpx::minmax_element</code>	Returns the smallest and the largest element in a range.	<hpx/algorithm.hpp>	<code>minmax_element</code> ¹⁰³

⁹⁸ http://en.cppreference.com/w/cpp/algorithm/is_heap⁹⁹ http://en.cppreference.com/w/cpp/algorithm/is_heap_until¹⁰⁰ http://en.cppreference.com/w/cpp/algorithm/make_heap¹⁰¹ http://en.cppreference.com/w/cpp/algorithm/max_element¹⁰² http://en.cppreference.com/w/cpp/algorithm/min_element¹⁰³ http://en.cppreference.com/w/cpp/algorithm/minmax_element

Table 2.18: Numeric Parallel Algorithms (In Header:
`<hpx/numeric.hpp>`)

Name	Description	In header	Algorithm page at cppreference.com
<code>hpx::adjacent_difference</code>	Calculates the difference between each element in an input range and the preceding element.	<code><hpx/numeric.hpp></code>	adjacent_difference ¹⁰⁴
<code>hpx::exclusive_scan</code>	Does an exclusive parallel scan over a range of elements.	<code><hpx/numeric.hpp></code>	exclusive_scan ¹⁰⁵
<code>hpx::reduce</code>	Sums up a range of elements.	<code><hpx/numeric.hpp></code>	reduce ¹⁰⁶
<code>hpx::inclusive_scan</code>	Does an inclusive parallel scan over a range of elements.	<code><hpx/algorithm.hpp></code>	inclusive_scan ¹⁰⁷
<code>hpx::parallel_reduce</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 <code>keys={1, 2, 3, 1}</code> , <code>values={9, 5, 30, 10}</code> .	<code><hpx/numeric.hpp></code>	
<code>hpx::transform_reduce</code>	Sums a range of elements after applying a function. Also, accumulates the inner products of two input ranges.	<code><hpx/numeric.hpp></code>	transform_reduce ¹⁰⁸
<code>hpx::transform_inclusive_scan</code>	Does an inclusive parallel scan over a range of elements after applying a function.	<code><hpx/numeric.hpp></code>	transform_inclusive_scan ¹⁰⁹
<code>hpx::parallel_exclusive_scan</code>	Does an exclusive parallel scan over a range of elements after applying a function.	<code><hpx/numeric.hpp></code>	transform_exclusive_scan ¹¹⁰

¹⁰⁴ http://en.cppreference.com/w/cpp/algorithm/adjacent_difference

¹⁰⁵ http://en.cppreference.com/w/cpp/algorithm/exclusive_scan

¹⁰⁶ <http://en.cppreference.com/w/cpp/algorithm/reduce>

¹⁰⁷ http://en.cppreference.com/w/cpp/algorithm/inclusive_scan

¹⁰⁸ http://en.cppreference.com/w/cpp/algorithm/transform_reduce

¹⁰⁹ http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan

¹¹⁰ http://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan

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

Name	Description	In header	Algorithm page at cppreference.com
<i>hpx::destroy</i>	Destroys a range of objects.	<i><hpx/memory.hpp></i>	destroy¹¹¹
<i>hpx::destroy_n</i>	Destroys a range of objects.	<i><hpx/memory.hpp></i>	destroy_n¹¹²
<i>hpx::uninitialized_copy</i>	Copies a range of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_copy¹¹³
<i>hpx::uninitialized_copy_n</i>	Copies a number of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_copy_n¹¹⁴
<i>hpx::uninitialized_default_construct</i>	Copies a range of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_default_construct¹¹⁵
<i>hpx::uninitialized_default_construct_n</i>	Copies as number of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_default_construct_n¹¹⁶
<i>hpx::uninitialized_fill</i>	Copies an object to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_fill¹¹⁷
<i>hpx::uninitialized_fill_n</i>	Copies an object to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_fill_n¹¹⁸
<i>hpx::uninitialized_move</i>	Moves a range of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_move¹¹⁹
<i>hpx::uninitialized_move_n</i>	Moves a number of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_move_n¹²⁰
<i>hpx::uninitialized_value_construct</i>	Constructs objects in an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_value_construct¹²¹
<i>hpx::uninitialized_value_construct_n</i>	Constructs objects in an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_value_construct_n¹²²

¹¹¹ <http://en.cppreference.com/w/cpp/memory/destroy>

¹¹² http://en.cppreference.com/w/cpp/memory/destroy_n

¹¹³ http://en.cppreference.com/w/cpp/memory/uninitialized_copy

¹¹⁴ http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n

¹¹⁵ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct

¹¹⁶ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n

¹¹⁷ http://en.cppreference.com/w/cpp/memory/uninitialized_fill

¹¹⁸ http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n

¹¹⁹ http://en.cppreference.com/w/cpp/memory/uninitialized_move

¹²⁰ http://en.cppreference.com/w/cpp/memory/uninitialized_move_n

¹²¹ http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct

¹²² http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct_n

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

Name	Description	In header
<code>hpx::for_loop</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/algorithm.hpp></code>
<code>hpx::for_loop_stride</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/algorithm.hpp></code>
<code>hpx::for_loop_n</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/algorithm.hpp></code>
<code>hpx::for_loop_n_stride</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/algorithm.hpp></code>

Executor parameters and executor parameter traits

HPX introduces the notion of execution parameters and execution parameter traits. At this point, the only parameter that can be customized is the size of the chunks of work executed on a single *HPX* thread (such as the number of loop iterations combined to run as a single task).

An executor parameter object is responsible for exposing the calculation of the size of the chunks scheduled. It abstracts the (potentially platform-specific) algorithms of determining those chunk sizes.

The way executor parameters are implemented is aligned with the way executors are implemented. All functionalities of concrete executor parameter types are exposed and accessible through a corresponding `hpx::parallel::executor_parameter_traits` type.

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

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

This call synchronously retrieves the size of a single chunk of loop iterations (or similar) to combine for execution on a single *HPX* thread if the overall number of tasks to schedule is given by `num_tasks`. The lambda function exposes a means of test-probing the execution of a single iteration for performance measurement purposes. The execution parameter type might dynamically determine the execution time of one or more tasks in order to calculate the chunk size; see `hpx::execution::auto_chunk_size` for an example of this executor parameter type.

Other functions in the interface exist to discover whether an executor parameter type should be invoked once (i.e., it returns a static chunk size; see `hpx::execution::static_chunk_size`) or whether it should be invoked for each scheduled chunk of work (i.e., it returns a variable chunk size; for an example, see `hpx::execution::guided_chunk_size`).

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

HPX implements the following executor parameter types:

- `hpx::execution::auto_chunk_size`: Loop iterations are divided into pieces and then assigned to threads. The number of loop iterations combined is determined based on measurements of how long the execution of 1% of the overall number of iterations takes. This executor parameter type makes sure that as many loop iterations are combined as necessary to run for the amount of time specified.
- `hpx::execution::static_chunk_size`: Loop iterations are divided into pieces of a given size and then assigned to threads. If the size is not specified, the iterations are, if possible, evenly divided contiguously among the threads. This executor parameters type is equivalent to OpenMP's STATIC scheduling directive.

- `hpx::execution::dynamic_chunk_size`: Loop iterations are divided into pieces of a given size and then dynamically scheduled among the cores; when a core finishes one chunk, it is dynamically assigned another. If the size is not specified, the default chunk size is 1. This executor parameter type is equivalent to OpenMP's DYNAMIC scheduling directive.
- `hpx::execution::guided_chunk_size`: Iterations are dynamically assigned to cores in blocks as cores request them until no blocks remain to be assigned. This is similar to `dynamic_chunk_size` except that the block size decreases each time a number of loop iterations is given to a thread. The size of the initial block is proportional to `number_of_iterations / number_of_cores`. Subsequent blocks are proportional to `number_of_iterations_remaining / number_of_cores`. The optional chunk size parameter defines the minimum block size. The default minimal chunk size is 1. This executor parameter type is equivalent to OpenMP's GUIDED scheduling directive.

Using task blocks

The `define_task_block`, `run` and the `wait` functions implemented based on N4411¹²³ are based on the `task_block` concept that is a part of the common subset of the Microsoft Parallel Patterns Library (PPL)¹²⁴ and the Intel Threading Building Blocks (TBB)¹²⁵ libraries.

These implementations adopt a simpler syntax than exposed by those libraries—one that is influenced by language-based concepts, such as `spawn` and `sync` from Cilk++¹²⁶ and `async` and `finish` from X10¹²⁷. They improve on existing practice in the following ways:

- The exception handling model is simplified and more consistent with normal C++ exceptions.
- Most violations of strict fork-join parallelism can be enforced at compile time (with compiler assistance, in some cases).
- The syntax allows scheduling approaches other than child stealing.

Consider an example of a parallel traversal of a tree, where a user-provided function `compute` is applied to each node of the tree, returning the sum of the results:

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

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

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

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

¹²³ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

¹²⁴ <https://msdn.microsoft.com/en-us/library/dd492418.aspx>

¹²⁵ <https://www.threadingbuildingblocks.org/>

¹²⁶ <https://software.intel.com/en-us/articles/intel-cilk-plus/>

¹²⁷ <https://x10-lang.org/>

unit of work that is allowed to execute in parallel with respect to the caller. Any parallel tasks spawned by `run` within the task block are joined back to a single thread of execution at the end of the `define_task_block`. `run` takes a user-provided function object `f` and starts it asynchronously—i.e., it may return before the execution of `f` completes. The *HPX* scheduler may choose to run `f` immediately or delay running `f` until compute resources become available.

A `task_block` can be constructed only by `define_task_block` because it has no public constructors. Thus, `run` can be invoked directly or indirectly only from a user-provided function passed to `define_task_block`:

```
void g();

void f(task_block<>& tr)
{
    tr.run(g);           // OK, invoked from within task_block in h
}

void h()
{
    define_task_block(f);
}

int main()
{
    task_block<> tr;   // Error: no public constructor
    tr.run(g);          // No way to call run outside of a define_task_block
    return 0;
}
```

Extensions for task blocks

Using execution policies with task blocks

HPX implements some extensions for `task_block` beyond the actual standards proposal N4411¹²⁸. The main addition is that a `task_block` can be invoked with an execution policy as its first argument, very similar to the parallel algorithms.

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

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

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

(continues on next page)

¹²⁸ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

(continued from previous page)

```

    return compute(n) + left + right;
}

```

This also causes the `hpx::parallel::v2::task_block` object to be a template in our implementation. The template argument is the type of the execution policy used to create the task block. The template argument defaults to `hpx::execution::parallel_policy`.

HPX still supports calling `hpx::parallel::v2::define_task_block` without an explicit execution policy. In this case the task block will run using the `hpx::execution::parallel_policy`.

HPX also adds the ability to access the execution policy that was used to create a given `task_block`.

Using executors to run tasks

Often, users want to be able to not only define an execution policy to use by default for all spawned tasks inside the task block, but also to customize the execution context for one of the tasks executed by `task_block::run`. Adding an optionally passed executor instance to that function enables this use case:

```

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

    define_task_block(
        execution::par,
        [&] (auto& tb) {
            if (n->left)
            {
                // use explicitly specified executor to run this task
                tb.run(my_executor(), [&] { left = traverse(n->left, compute); });
            }
            if (n->right)
            {
                // use the executor associated with the par execution policy
                tb.run([&] { right = traverse(n->right, compute); });
            }
        });
    return compute(n) + left + right;
}

```

HPX still supports calling `hpx::parallel::v2::task_block::run` without an explicit executor object. In this case the task will be run using the executor associated with the execution policy that was used to call `hpx::parallel::v2::define_task_block`.

2.3.6 Writing distributed HPX applications

This section focuses on the features of *HPX* needed to write distributed applications, namely the *Active Global Address Space* (*AGAS*), remotely executable functions (i.e. *actions*), and distributed objects (i.e. *components*).

Global names

HPX implements an *Active Global Address Space* (*AGAS*) which is exposing a single uniform address space spanning all localities an application runs on. *AGAS* is a fundamental component of the ParalleX execution model. Conceptually, there is no rigid demarcation of local or global memory in *AGAS*; all available memory is a part of the same address space. *AGAS* enables named objects to be moved (migrated) across localities without having to change the object's name, i.e., no references to migrated objects have to be ever updated. This feature has significance for dynamic load balancing and in applications where the workflow is highly dynamic, allowing work to be migrated from heavily loaded nodes to less loaded nodes. In addition, immutability of names ensures that *AGAS* does not have to keep extra indirections ("bread crumbs") when objects move, hence minimizing complexity of code management for system developers as well as minimizing overheads in maintaining and managing aliases.

The *AGAS* implementation in *HPX* does not automatically expose every local address to the global address space. It is the responsibility of the programmer to explicitly define which of the objects have to be globally visible and which of the objects are purely local.

In *HPX* global addresses (global names) are represented using the `hpx::id_type` data type. This data type is conceptually very similar to `void*` pointers as it does not expose any type information of the object it is referring to.

The only predefined global addresses are assigned to all localities. The following *HPX* API functions allow one to retrieve the global addresses of localities:

- `hpx::find_here`: retrieve the global address of the *locality* this function is called on.
- `hpx::find_all_localities`: retrieve the global addresses of all localities available to this application (including the *locality* the function is being called on).
- `hpx::find_remote_localities`: retrieve the global addresses of all remote localities available to this application (not including the *locality* the function is being called on)
- `hpx::get_num_localities`: retrieve the number of localities available to this application.
- `hpx::find_locality`: retrieve the global address of any *locality* supporting the given component type.
- `hpx::get_colocation_id`: retrieve the global address of the *locality* currently hosting the object with the given global address.

Additionally, the global addresses of localities can be used to create new instances of components using the following *HPX* API function:

- `hpx::components::new_`: Create a new instance of the given Component type on the specified *locality*.

Note: *HPX* does not expose any functionality to delete component instances. All global addresses (as represented using `hpx::id_type`) are automatically garbage collected. When the last (global) reference to a particular component instance goes out of scope the corresponding component instance is automatically deleted.

Applying actions

Action type definition

Actions are special types we use to describe possibly remote operations. For every global function and every member function which has to be invoked distantly, a special type must be defined. For any global function the special macro `HPX_PLAIN_ACTION` can be used to define the action type. Here is an example demonstrating this:

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }
}

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

Important: The macro `HPX_PLAIN_ACTION` has to be placed in global namespace, even if the wrapped function is located in some other namespace. The newly defined action type is placed in the global namespace as well.

If the action type should be defined somewhere not in global namespace, the action type definition has to be split into two macro invocations (`HPX_DEFINE_PLAIN_ACTION` and `HPX_REGISTER_ACTION`) as shown in the next example:

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }

    // On conforming compilers the following macro expands to:
    //
    //     typedef hpx::actions::make_action<
    //         decltype(&some_global_function), &some_global_function
    //     >::type some_global_action;
    //
    // This will define the action type 'some_global_action' which represents
    // the function 'some_global_function'.
    HPX_DEFINE_PLAIN_ACTION(some_global_function, some_global_action);
}

// The following macro expands to a series of definitions of global objects
// which are needed for proper serialization and initialization support
// enabling the remote invocation of the function ``some_global_function``.
HPX_REGISTER_ACTION(app::some_global_action, app_some_global_action);
```

The shown code defines an action type `some_global_action` inside the namespace `app`.

Important: If the action type definition is split between two macros as shown above, the name of the action type to create has to be the same for both macro invocations (here `some_global_action`).

Important: The second argument passed to `HPX_REGISTER_ACTION` (`app_some_global_action`) has to comprise a globally unique C++ identifier representing the action. This is used for serialization purposes.

For member functions of objects which have been registered with AGAS (e.g. ‘components’) a different registration macro `HPX_DEFINE_COMPONENT_ACTION` has to be utilized. Any component needs to be declared in a header file and have some special support macros defined in a source file. Here is an example demonstrating this. The first snippet has to go into the header file:

```
namespace app
{
    struct some_component
        : hpx::components::component_base<some_component>
    {
        int some_member_function(std::string s)
        {
            return boost::lexical_cast<int>(s);
        }

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

// Note: The second argument to the macro below has to be systemwide-unique
//       C++ identifiers
HPX_REGISTER_ACTION_DECLARATION(app::some_component::some_member_action, some_
                                component_some_action);
```

The next snippet belongs into a source file (e.g. the main application source file) in the simplest case:

```
typedef hpx::components::component<app::some_component> component_type;
typedef app::some_component some_component;

HPX_REGISTER_COMPONENT(component_type, some_component);

// The parameters for this macro have to be the same as used in the corresponding
// HPX_REGISTER_ACTION_DECLARATION() macro invocation above
typedef some_component::some_member_action some_component_some_action;
HPX_REGISTER_ACTION(some_component_some_action);
```

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

The most important macro invocation is the `HPX_DEFINE_COMPONENT_ACTION` in the header file as this defines the action type we need to invoke the member function. For a complete example of a simple component action see `component_in_executable.cpp`.

Action invocation

The process of invoking a global function (or a member function of an object) with the help of the associated action is called ‘applying the action’. Actions can have arguments, which will be supplied while the action is applied. At the minimum, one parameter is required to apply any action - the id of the *locality* the associated function should be invoked on (for global functions), or the id of the component instance (for member functions). Generally, *HPX* provides several ways to apply an action, all of which are described in the following sections.

Generally, *HPX* actions are very similar to ‘normal’ C++ functions except that actions can be invoked remotely. Fig. ?? below shows an overview of the main API exposed by *HPX*. This shows the function invocation syntax as defined by the C++ language (dark gray), the additional invocation syntax as provided through C++ Standard Library features (medium gray), and the extensions added by *HPX* (light gray) where:

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

<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>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 meantime, allowing the application to make further progress while the remote result is computed. This helps overlapping computation with communication and hiding communication latencies.

Note: The syntax of applying an action is always the same, regardless whether the target *locality* is remote to the invocation *locality* or not. This is a very important feature of *HPX* as it frees the user from the task of keeping track what actions have to be applied locally and which actions are remote. If the target for applying an action is local, a new thread is automatically created and scheduled. Once this thread is scheduled and run, it will execute the function encapsulated by that action. If the target is remote, *HPX* will send a parcel to the remote *locality* which encapsulates the action and its parameters. Once the parcel is received on the remote *locality* *HPX* will create and schedule a new thread there. Once this thread runs on the remote *locality*, it will execute the function encapsulated by the action.

Applying an action with a continuation but without any synchronization

This method is very similar to the method described in section *Applying an action asynchronously without any synchronization*. The difference is that it allows the user to chain a sequence of asynchronous operations, while handing the (intermediate) results from one step to the next step in the chain. Where `hpx::apply` invokes a single function using ‘fire and forget’ semantics, `hpx::apply_continue` triggers a chain of functions without the need for the execution flow ‘to come back’ to the invocation site. Each of the asynchronous functions can be executed on a different *locality*.

Applying an action with a continuation and with synchronization

This method is very similar to the method described in section *Applying an action asynchronously with synchronization*. In addition to what `hpx::async` can do, the functions `hpx::async_continue` takes an additional function argument. This function will be called as the continuation of the executed action. It is expected to perform additional operations and to make sure that a result is returned to the original invocation site. This method chains operations asynchronously by providing a continuation operation which is automatically executed once the first action has finished executing.

As an example we chain two actions, where the result of the first action is forwarded to the second action and the result of the second action is sent back to the original invocation site:

```
// first action
std::int32_t action1(std::int32_t i)
{
    return i+1;
}
HPX_PLAIN_ACTION(action1);      // defines action1_type

// second action
std::int32_t action2(std::int32_t i)
{
    return i*2;
}
HPX_PLAIN_ACTION(action2);      // defines action2_type

// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;           // define an instance of 'action1_type'
action2_type act2;           // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1, hpx::make_continuation(act2),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n"; // will print: 86 ((42 + 1) * 2)
```

By default, the continuation is executed on the same *locality* as `hpx::async_continue` is invoked from. If you want to specify the *locality* where the continuation should be executed, the code above has to be written as:

```
// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;           // define an instance of 'action1_type'
action2_type act2;           // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1, hpx::make_continuation(act2, hpx::find_here())),
```

(continues on next page)

(continued from previous page)

```
    hpx::find_here(), 42);
hpx::cout << f.get() << "\n"; // will print: 86 ((42 + 1) * 2)
```

Similarly, it is possible to chain more than 2 operations:

```
action1_type act1; // define an instance of 'action1_type'
action2_type act2; // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1,
        hpx::make_continuation(act2, hpx::make_continuation(act1)),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n"; // will print: 87 ((42 + 1) * 2 + 1)
```

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

```
struct continuation
{
    template <typename Result>
    void operator()(hpx::id_type id, Result&& result) const
    {
        ...
    }
};
```

where the parameters passed to the overloaded function `operator()` 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_distributed/find_localities.hpp>

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

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

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

(continues on next page)

(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 2.21: Sequence containers

Name	Description	In header	Class page at cppreference.com
hpx::partitioned_	Dynamic segmented contiguous array.	<hpx/include/partitioned_vector.hpp>	vector ¹²⁹

Table 2.22: Unordered associative containers

Name	Description	In header	Class page at cppreference.com
hpx::unordered_map ¹²⁹	Segmented collection of key-value pairs, hashed by keys, keys are unique.	<hpx/include/unordered_map.hpp>	unordered_map ¹³⁰

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)

¹²⁹ <http://en.cppreference.com/w/cpp/container/vector>

¹³⁰ http://en.cppreference.com/w/cpp/container/unordered_map

(continued from previous page)

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

Which is equivalent to:

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

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

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

Using views

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

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

Preface: Why SPMD?

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

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

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

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */)
{
    // Parallel section

    /* Do some code */
}
HPC_PLAIN_ACTION(bulk_function, bulk_action);
```

Note: In the following paragraphs, we will use the term “image” several times. An image is defined as a lightweight process whose entry point is a function provided by the user. It’s an “image of the function”.

The `spmd_block` class contains the following methods:

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

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

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

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */)
{
    std::size_t num_images = block.get_num_images();
    std::size_t this_image = block.this_image();
    std::size_t images_per_locality = block.images_per_locality();

    /* Do some code */

    // Synchronize all images in the team
    block.sync_all();

    /* Do some code */

    // Synchronize image 0 and image 1
    block.sync_images(0,1);

    /* Do some code */

    std::vector<std::size_t> vec_images = {2,3,4};

    // Synchronize images 2, 3 and 4
    block.sync_images(vec_images);

    // Alternative call to synchronize images 2, 3 and 4
    block.sync_images(vec_images.begin(), vec_images.end());

    /* Do some code */

    // Non-blocking version of sync_all()
    hpx::future<void> event =
        block.sync_all(hpx::launch::async);

    // Callback waiting for 'event' to be ready before being scheduled
    hpx::future<void> cb =
        event.then(
            [] (hpx::future<void>)
            {

                /* Do some code */

            });
    // Finally wait for the execution tree to be finished
    cb.get();
}

HPX_PLAIN_ACTION(bulk_test_function, bulk_test_action);
```

Then, in order to invoke the parallel section, call the function `define_spmd_block` specifying an arbitrary symbolic name and indicating the number of images per *locality* to create:

```

void bulk_function(hpx::lcos::spmd_block block, /* , arg0, arg1, ... */)
{
}

HPX_PLAIN_ACTION(bulk_test_function, bulk_test_action);

int main()
{
    /* std::size_t arg0, arg1, ... */

    bulk_action act;
    std::size_t images_per_locality = 4;

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

    return 0;
}

```

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

SPMD multidimensional views

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

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

```

#include <hpx/include/partitioned_vector.hpp>

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

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

hpx::partitioned_vector<double> v;

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

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

```

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

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

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

Subscript-based operations

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

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

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

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

/* Do some code */

Vec v;

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

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

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

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

Iterator-based operations

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

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

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

(continues on next page)

(continued from previous page)

```

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

/* Do some code */

Vec v1, v2;

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

    // Instantiate 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 = local_vv.begin(); i != local_vv.end(); i++)
    {
        std::vector<float> & segment = *i;

        /* Do some code */
    }
}
}

```

Instantiating sub-views

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

```

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

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

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

/* Do some code */

Vec v;

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

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

    // Instantiate the subview
    View_2D svv(

```

(continues on next page)

(continued from previous page)

```

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

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

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

}

```

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

C++ co-arrays

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

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

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

```

#include <hpx/components/containers/coarray/coarray.hpp>
#include <hpx/collectives/spmd_block.hpp>

// The following code generates all necessary boiler plate to enable the
// co-creation of 'coarray'
//
HPX_REGISTER_COARRAY(double);

// Parallel section (suppose 'block' an spmd_block instance)
{
    using hpx::container::placeholders::_;

    std::size_t height=32, width=4, segment_size=10;

    hpx::coarray<double, 3> a(block, "a", {height, width, _}, segment_size);
    /* Do some code */
}

```

Unlike segmented containers, a co-array object can only be instantiated within a parallel section. Here is the description of the parameters to provide to the coarray constructor:

Table 2.23: Parameters of coarray constructor

Parameter	Description
block	Reference to a <code>spmd_block</code> object
"a"	Symbolic name of type <code>std::string</code>
{height, width, _}	Dimensions of the coarray object
segment_size	Size of a co-indexed element (i.e. size of the object referenced by the expression <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/collectives/spmd_block.hpp>

// The following code generates all necessary boiler plate to enable the
// co-creation of 'coarray'
//
HPX_REGISTER_COARRAY(double);

// Parallel section (suppose 'block' an spmd_block instance)
{
    using hpx::container::placeholders::_;

    std::size_t height=32, width=4, segment_size=10;

    hpx::coarray<double, 3> a(block, "a", {height, width, _}, segment_size);

    double idx = block.this_image()*height*width;

    for (std::size_t j = 0; j<width; j++)
        for (std::size_t i = 0; i<height; i++)

```

(continues on next page)

(continued from previous page)

```

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

block.sync_all();
}

```

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

2.3.7 Running on batch systems

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

How to use *HPX* applications with PBS

Most *HPX* applications are executed on parallel computers. These platforms typically provide integrated job management services that facilitate the allocation of computing resources for each parallel program. *HPX* includes support for one of the most common job management systems, the Portable Batch System (PBS).

All PBS jobs require a script to specify the resource requirements and other parameters associated with a parallel job. The PBS script is basically a shell script with PBS directives placed within commented sections at the beginning of the file. The remaining (not commented-out) portions of the file executes just like any other regular shell script. While the description of all available PBS options is outside the scope of this tutorial (the interested reader may refer to in-depth documentation¹³¹ for more information), below is a minimal example to illustrate the approach. The following test application will use the multithreaded `hello_world_distributed` program, explained in the section *Remote execution with actions*.

```

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

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

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

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

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

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

¹³¹ <http://www.clusterresources.com/torquedocs21/>

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

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

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

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

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

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

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

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

Another choice is for the `pbsdsh` command in your main job script to invoke your program via a shell, like `sh` or `bash`, so that it gives an initialized environment for each instance. Users can create a small script `runme.sh`, which is used to invoke the program:

```
#!/bin/bash
# Small script which invokes the program based on what was passed on its
# command line.
#
# This script is executed by the bash shell which will initialize all
# environment variables as usual.
$@
```

Now, the script is invoked using the `pbsdsh` tool:

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

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

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

All that remains now is submitting the job to the queuing system. Assuming that the contents of the PBS script were saved in the file `pbs_hello_world.sh` in the current directory, this is accomplished by typing:

```
$ qsub ./pbs_hello_world_pbs.sh
```

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

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

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

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

and look for a single-letter job status symbol. The common cases include:

- *Q* - signifies that the job is queued and awaiting its turn to be executed.
- *R* - indicates that the job is currently running.
- *C* - means that the job has completed.

The example qstat output below shows a job waiting for execution resources to become available:

Job id	Name	User	Time Use S Queue
42.supercomputer	...ello_world.sh	joe_user	0 Q batch

After the job completes, PBS will place two files, `pbs_hello_world.sh.o42` and `pbs_hello_world.sh.e42`, in the directory where the job was submitted. The first contains the standard output and the second contains the standard error from all the nodes on which the application executed. In our example, the error output file should be empty and the standard output file should contain something similar to:

```
hello world from OS-thread 3 on locality 0
hello world from OS-thread 2 on locality 0
hello world from OS-thread 1 on locality 1
hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 1
hello world from OS-thread 2 on locality 1
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 1
```

Congratulations! You have just run your first distributed *HPX* application!

How to use *HPX* applications with SLURM

Just like PBS (described in section *How to use HPX applications with PBS*), SLURM is a job management system which is widely used on large supercomputing systems. Any *HPX* application can easily be run using SLURM. This section describes how this can be done.

The easiest way to run an *HPX* application using SLURM is to utilize the command line tool `srun`, which interacts with the SLURM batch scheduling system:

```
$ srun -p <partition> -N <number-of-nodes> hpx-application <application-arguments>
```

Here, `<partition>` is one of the node partitions existing on the target machine (consult the machine's documentation to get a list of existing partitions) and `<number-of-nodes>` is the number of compute nodes that should be used. By default, the *HPX* application is started with one *locality* per node and uses all available cores on a node. You can change the number of localities started per node (for example, to account for NUMA effects) by specifying the `-n` option of `srun`. The number of cores per *locality* can be set by `-c`. The `<application-arguments>` are any application specific arguments that need to be passed on to the application.

Note: There is no need to use any of the *HPX* command line options related to the number of localities, number of threads, or related to networking ports. All of this information is automatically extracted from the SLURM environment by the *HPX* startup code.

Important: The `srun` documentation explicitly states: “If `-c` is specified without `-n`, as many tasks will be allocated per node as possible while satisfying the `-c` restriction. For instance on a cluster with 8 CPUs per node, a job request for 4 nodes and 3 CPUs per task may be allocated 3 or 6 CPUs per node (1 or 2 tasks per node) depending upon resource consumption by other jobs.” For this reason, it’s recommended to always specify `-n <number-of-instances>`, even if `<number-of-instances>` is equal to one (1).

Interactive shells

To get an interactive development shell on one of the nodes, users can issue the following command:

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

After the shell has been opened, users can run their *HPX* application. By default, it uses all available cores. Note that if you requested one node, you don’t need to do `srun` again. However, if you requested more than one node, and want to run your distributed application, you can use `srun` again to start up the distributed *HPX* application. It will use the resources that have been requested for the interactive shell.

Scheduling batch jobs

The above mentioned method of running *HPX* applications is fine for development purposes. The disadvantage that comes with `srun` is that it only returns once the application is finished. This might not be appropriate for longer-running applications (for example, benchmarks or larger scale simulations). In order to cope with that limitation, users can use the `sbatch` command.

The `sbatch` command expects a script that it can run once the requested resources are available. In order to request resources, users need to add `#SBATCH` comments in their script or provide the necessary parameters to `sbatch` directly. The parameters are the same as with `run`. The commands you need to execute are the same you would need to start your application as if you were in an interactive shell.

2.3.8 Debugging *HPX* applications

Using a debugger with *HPX* applications

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

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

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

additional threads (used for service thread pools). The first worker thread is usually the eighth thread, and most user codes will be run on these worker threads. The last thread is a helper thread used for *HPX* shutdown.

Finally, since *HPX* is a multi-threaded runtime, the following *gdb* options can be helpful:

```
set pagination off
set non-stop on
```

Non-stop mode allows users to have a single thread stop on a breakpoint without stopping all other threads as well.

Using sanitizers with *HPX* applications

Warning: Not all parts of *HPX* are sanitizer clean. This means that users may end up with false positives from *HPX* itself when using sanitizers for their applications.

To use sanitizers with *HPX*, turn on `HPX_WITH_SANITIZERS` and turn off `HPX_WITH_STACKOVERFLOW_DETECTION` during CMake configuration. It's recommended to also build Boost with the same sanitizers that will be used for *HPX*. The appropriate sanitizers can then be enabled using CMake by appending `-fsanitize=address -fno-omit-frame-pointer` to `CMAKE_CXX_FLAGS` and `-fsanitize=address` to `CMAKE_EXE_LINKER_FLAGS`. Replace `address` with the sanitizer that you want to use.

Debugging applications using core files

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

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

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

```
$ ulimit -c unlimited
```

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

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

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

2.3.9 Optimizing HPX applications

Performance counters

Performance counters in *HPX* are used to provide information as to how well the runtime system or an application is performing. The counter data can help determine system bottlenecks, and fine-tune system and application performance. The *HPX* runtime system, its networking, and other layers provide counter data that an application can consume to provide users with information about how well the application is performing.

Applications can also use counter data to determine how much system resources to consume. For example, an application that transfers data over the network could consume counter data from a network switch to determine how much data to transfer without competing for network bandwidth with other network traffic. The application could use the counter data to adjust its transfer rate as the bandwidth usage from other network traffic increases or decreases.

Performance counters are *HPX* parallel processes that expose a predefined interface. *HPX* exposes special API functions that allow one to create, manage, and read the counter data, and release instances of performance counters. Performance Counter instances are accessed by name, and these names have a predefined structure which is described in the section *Performance counter names*. The advantage of this is that any Performance Counter can be accessed remotely (from a different *locality*) or locally (from the same *locality*). Moreover, since all counters expose their data using the same API, any code consuming counter data can be utilized to access arbitrary system information with minimal effort.

Counter data may be accessed in real time. More information about how to consume counter data can be found in the section *Consuming performance counter data*.

All *HPX* applications provide command line options related to performance counters, such as the ability to list available counter types, or periodically query specific counters to be printed to the screen or save them in a file. For more information, please refer to the section *HPX Command Line Options*.

Performance counter names

All Performance Counter instances have a name uniquely identifying each instance. This name can be used to access the counter, retrieve all related meta data, and to query the counter data (as described in the section *Consuming performance counter data*). Counter names are strings with a predefined structure. The general form of a countername is:

```
/objectname{full_instancename}/countername@parameters
```

where `full_instancename` could be either another (full) counter name or a string formatted as:

```
parentinstancename#parentindex/instancename#instanceindex
```

Each separate part of a countername (e.g., `objectname`, `countername`, `parentinstancename`, `instancename`, and `parameters`) should start with a letter ('a'...'z', 'A'...'Z') or an underscore character ('_'), optionally followed by letters, digits ('0'...'9'), hyphen ('-'), or underscore characters. Whitespace is not allowed inside a counter name. The characters '/', '{', '}', '#', and '@' have a special meaning and are used to delimit the different parts of the counter name.

The parts `parentinstanceindex` and `instanceindex` are integers. If an index is not specified, *HPX* will assume a default of -1.

Two counter name examples

This section gives examples of both simple counter names and aggregate counter names. For more information on simple and aggregate counter names, please see *Performance counter instances*.

An example of a well-formed (and meaningful) simple counter name would be:

```
/threads{locality#0/total}/count/cumulative
```

This counter returns the current cumulative number of executed (retired) HPX threads for the *locality* 0. The counter type of this counter is /threads/count/cumulative and the full instance name is locality#0/total. This counter type does not require an `instanceindex` or `parameters` to be specified.

In this case, the `parentindex` (the '0') designates the *locality* for which the counter instance is created. The counter will return the number of HPX threads retired on that particular *locality*.

Another example for a well formed (aggregate) counter name is:

```
/statistics{/threads{locality#0/total}/count/cumulative}/average@500
```

This counter takes the simple counter from the first example, samples its values every 500 milliseconds, and returns the average of the value samples whenever it is queried. The counter type of this counter is /statistics/average and the instance name is the full name of the counter for which the values have to be averaged. In this case, the `parameters` (the '500') specify the sampling interval for the averaging to take place (in milliseconds).

Performance counter types

Every performance counter belongs to a specific performance counter type which classifies the counters into groups of common semantics. The type of a counter is identified by the `objectname` and the `countername` parts of the name.

```
/objectname/countername
```

When an application starts HPX will register all available counter types on each of the localities. These counter types are held in a special performance counter registration database, which can be used to retrieve the meta data related to a counter type and to create counter instances based on a given counter instance name.

Performance counter instances

The `full_instancename` distinguishes different counter instances of the same counter type. The formatting of the `full_instancename` depends on the counter type. There are two types of counters: simple counters, which usually generate the counter values based on direct measurements, and aggregate counters, which take another counter and transform its values before generating their own counter values. An example for a simple counter is given *above*: counting retired HPX threads. An aggregate counter is shown as an example *above* as well: calculating the average of the underlying counter values sampled at constant time intervals.

While simple counters use instance names formatted as `parentinstancename#parentindex/instancename#instanceindex`, most aggregate counters have the full counter name of the embedded counter as their instance name.

Not all simple counter types require specifying all four elements of a full counter instance name; some of the parts (`parentinstancename`, `parentindex`, `instancename`, and `instanceindex`) are optional for specific counters. Please refer to the documentation of a particular counter for more information about the formatting requirements for the name of this counter (see *Existing HPX performance counters*).

The parameters are used to pass additional information to a counter at creation time. They are optional, and they fully depend on the concrete counter. Even if a specific counter type allows additional parameters to be given, those usually are not required as sensible defaults will be chosen. Please refer to the documentation of a particular counter for more information about what parameters are supported, how to specify them, and what default values are assumed (see also *Existing HPX performance counters*).

Every *locality* of an application exposes its own set of performance counter types and performance counter instances. The set of exposed counters is determined dynamically at application start based on the execution environment of the application. For instance, this set is influenced by the current hardware environment for the *locality* (such as whether the *locality* has access to accelerators), and the software environment of the application (such as the number of OS threads used to execute *HPX* threads).

Using wildcards in performance counter names

It is possible to use wildcard characters when specifying performance counter names. Performance counter names can contain two types of wildcard characters:

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

A wildcard character has a meaning which is very close to usual file name wildcard matching rules implemented by common shells (like bash).

Table 2.24: 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 2.25: 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, the *HPX* application or the *HPX* API. The command line interface is easier to use, but it is less flexible and does not allow one to adjust the behaviour of your application at runtime. The command line interface provides a convenience abstraction but simplified abstraction for querying and logging performance counter data for a set of performance counters.

Consuming performance counter data from the command line

HPX provides a set of predefined command line options for every application that uses `hpx::init` for its initialization. While there are many more command line options available (see *HPX Command Line Options*), the set of options related to performance counters allows one to list existing counters, and query existing counters once at application termination or repeatedly after a constant time interval.

The following table summarizes the available command line options:

Table 2.26: HPX Command Line Options Related to Performance Counters

Com-mand line option	Description
<code>--hpx:print</code>	Prints the specified performance counter either repeatedly and/or at the times specified by <code>--hpx:print-counter-at</code> (see also option <code>--hpx:print-counter-interval</code>).
<code>--hpx:print-counter-at</code>	Prints the specified performance counter either repeatedly and/or at the times specified by <code>--hpx:print-counter-at</code> . Reset the counter after the value is queried (see also option <code>--hpx:print-counter-interval</code>).
<code>--hpx:print-counter-interval</code>	Prints the performance counter(s) specified with <code>--hpx:print-counter</code> repeatedly after the time interval (specified in milliseconds) (default: 0 which means print once at shutdown).
<code>--hpx:print-counter-file</code>	Prints the performance counter(s) specified with <code>--hpx:print-counter</code> to the given file (default: console).
<code>--hpx:list</code>	Lists the names of all registered performance counters.
<code>--hpx:list-info</code>	Lists the description of all registered performance counters.
<code>--hpx:print-format</code>	Prints 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: csv (prints counter values in CSV format with full names as header) csv-short (prints counter values in CSV format with shortnames provided with <code>--hpx:print-counter</code> as <code>--hpx:print-counter shortname,full-countername</code>).
<code>--hpx:no-csv-header</code>	Prints the performance counter(s) specified with <code>--hpx:print-counter</code> and csv or csv-short format specified with <code>--hpx:print-counter-format</code> without header.
<code>--hpx:print-counter-reset</code>	Prints the performance counter(s) specified with <code>--hpx:print-counter</code> (or <code>--hpx:print-counter-reset</code>) at the given point in time. Possible argument values: startup, shutdown (default), nosutdown.
<code>--hpx:reset</code>	Resets all performance counter(s) specified with <code>--hpx:print-counter</code> after they have been evaluated.
<code>--hpx:print-type</code>	Appends counter type description to generated output.
<code>--hpx:print-locality</code>	Each locality prints only its own local counters.

While the options `--hpx:list-counters` and `--hpx:list-counter-infos` give a short list of all available counters, the full documentation for those can be found in the section *Existing HPX performance counters*.

A simple example

All of the commandline options mentioned above can be tested using the `hello_world_distributed` example.

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

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

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

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

fullname: /agas/count/bind helptext: returns the number of invocations of the AGAS service 'bind' type: counter_raw version: 1.0.0
-----

fullname: /agas/count/bind_gid helptext: returns the number of invocations of the AGAS service 'bind_gid' type: counter_raw version: 1.0.0
-----

...

```

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

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

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

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

which yields for instance:

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

The first line is the normal output generated by `hello_world_distributed` and has no relation to the counter data listed. The last two lines contain the counter data as gathered at application shutdown. These lines have six

fields, the counter name, the sequence number of the counter invocation, the time stamp at which this information has been sampled, the unit of measure for the time stamp, the actual counter value and an optional unit of measure for the counter value.

Note: The command line option `--hpx:print-counter-types` will append a seventh field to the generated output. This field will hold an abbreviated counter type.

The actual counter value can be represented by a single number (for counters returning singular values) or a list of numbers separated by '`:`' (for counters returning an array of values, like for instance a histogram).

Note: The name of the performance counter will be enclosed in double quotes '`"`' if it contains one or more commas '`,`'.

Requesting to query the counter data once after a constant time interval with this command line:

```
$ hello_world_distributed \
  --hpx:print-counter=/threads{locality#0/total}/count/cumulative \
  --hpx:print-counter=/agas{locality#0/total}/count/bind \
  --hpx:print-counter-interval=20
```

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

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

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

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

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

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

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

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

With format as csv:

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

will print the values of performance counters in CSV format with the full countername as a header:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
/threads{locality#/*/total}/count/cumulative,/threads{locality#/*/total}/count/
↪cumulative-phases
39,93
```

With format csv-short:

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

will print the values of performance counters in CSV format with the short countername as a header:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
cumulative,phases
39,93
```

With format csv and csv-short when used with --hpx:print-counter-interval:

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

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

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

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

```
$ hello_world_distributed \
--hpx:threads 2 \
--hpx:print-counter-format csv-short \
--hpx:print-counter cumulative,/threads{locality#/*/total}/count/cumulative \
--hpx:print-counter phases,/threads{locality#/*/total}/count/cumulative-phases \
--hpx:no-csv-header
```

will print:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
37,91
```

Consuming performance counter data using the *HPX API*

HPX provides an API that allows users to discover performance counters and to retrieve the current value of any existing performance counter from any application.

Discover existing performance counters

Retrieve the current value of any performance counter

Performance counters are specialized *HPX* components. In order to retrieve a counter value, the performance counter needs to be instantiated. *HPX* exposes a client component object for this purpose:

```
hpx::performance_counters::performance_counter counter(std::string const& name);
```

Instantiating an instance of this type will create the performance counter identified by the given name. Only the first invocation for any given counter name will create a new instance of that counter. All following invocations for a given counter name will reference the initially created instance. This ensures that at any point in time there is never more than one active instance of any of the existing performance counters.

In order to access the counter value (or to invoke any of the other functionality related to a performance counter, like start, stop or reset) member functions of the created client component instance should be called:

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

For more information about the client component type, see `hpx::performance_counters::performance_counter`

Note: In the above example `count.get_value()` returns a future. In order to print the result we must append `.get()` to retrieve the value. You could write the above example like this for more clarity:

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

Providing performance counter data

HPX offers several ways by which you may provide your own data as a performance counter. This has the benefit of exposing additional, possibly application-specific information using the existing Performance Counter framework, unifying the process of gathering data about your application.

An application that wants to provide counter data can implement a performance counter to provide the data. When a consumer queries performance data, the HPX runtime system calls the provider to collect the data. The runtime system uses an internal registry to determine which provider to call.

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

Exposing performance counter data using a simple function

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

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

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

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

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

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

This example function exposes a linearly-increasing value as our performance data. The value is incremented on each invocation, i.e., each time a consumer requests the counter data of this performance counter.

The next step in exposing this counter to the runtime system is to register the function as a new raw counter type using the HPX API function `hpx::performance_counters::install_counter_type`. A counter type represents certain common characteristics of counters, like their counter type name and any associated description information. The following snippet shows an example of how to register the function `some_performance_data`, which is shown above, for a counter type named "/test/data". This registration has to be executed before any consumer instantiates, and queries an instance of this counter type:

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

void register_counter_type()
{
    // Call the HPX API function to register the counter type.
    hpx::performance_counters::install_counter_type(
        "/test/data",                                         // counter type name
        &some_performance_data,                            // function providing counter_
        &data);
```

(continues on next page)

(continued from previous page)

```

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

```

Now it is possible to instantiate a new counter instance based on the naming scheme `"/test{locality#*/total}/data"` where `*` is a zero-based integer index identifying the *locality* for which the counter instance should be accessed. The function `hpx::performance_counters::install_counter_type` enables users to instantiate exactly one counter instance for each *locality*. Repeated requests to instantiate such a counter will return the same instance, i.e., the instance created for the first request.

If this counter needs to be accessed using the standard *HPX* command line options, the registration has to be performed during application startup, before `hpx_main` is executed. The best way to achieve this is to register an *HPX* startup function using the API function `hpx::register_startup_function` before calling `hpx::init` to initialize the runtime system:

```

int main(int argc, char* argv[])
{
    // By registering the counter type we make it available to any consumer
    // who creates and queries an instance of the type "/test/data".
    //
    // This registration should be performed during startup. The
    // function 'register_counter_type' should be executed as an HPX thread right
    // before hpx_main is executed.
    hpx::register_startup_function(&register_counter_type);

    // Initialize and run HPX.
    return hpx::init(argc, argv);
}

```

Please see the code in `simplest_performance_counter.cpp` for a full example demonstrating this functionality.

Implementing a full performance counter

Sometimes, the simple way of exposing a single value as a performance counter is not sufficient. For that reason, *HPX* provides a means of implementing full performance counters which support:

- Retrieving the descriptive information about the performance counter
- Retrieving the current counter value
- Resetting the performance counter (value)
- Starting the performance counter
- Stopping the performance counter
- Setting the (initial) value of the performance counter

Every full performance counter will implement a predefined interface:

```

// Copyright (c) 2007-2020 Hartmut Kaiser
//
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

```

(continues on next page)

(continued from previous page)

```

#pragma once

#include <hpx/config.hpp>
#include <hpx/async_base/launch_policy.hpp>
#include <hpx/components/client_base.hpp>
#include <hpx/functional/bind_front.hpp>
#include <hpx/futures/future.hpp>
#include <hpx/modules/execution.hpp>

#include <hpx/performance_counters/counters_fwd.hpp>
#include <hpx/performance_counters/server/base_performance_counter.hpp>

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

///////////////////////////////
namespace hpx { namespace performance_counters {

/////////////////////////////
struct HPX_EXPORT performance_counter
    : components::client_base<performance_counter,
      server::base_performance_counter>
{
    using base_type = components::client_base<performance_counter,
      server::base_performance_counter>;

    performance_counter() = default;

    performance_counter(std::string const& name);

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

    performance_counter(id_type const& id)
        : base_type(id)
    {
    }

    performance_counter(future<id_type>&& id)
        : base_type(HPX_MOVE(id))
    {
    }

    performance_counter(hpx::future<performance_counter>&& c)
        : base_type(HPX_MOVE(c))
    {
    }

    /////////////////////////
    future<counter_info> get_info() const;
    counter_info get_info(
        launch::sync_policy, error_code& ec = throws) const;

    future<counter_value> get_counter_value(bool reset = false);
    counter_value get_counter_value(

```

(continues on next page)

(continued from previous page)

```

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

future<counter_value> get_counter_value() const;
counter_value get_counter_value(
    launch::sync_policy, error_code& ec = throws) const;

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

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

```

(continues on next page)

(continued from previous page)

```

    {
        return get_counter_value().then(hpx::launch::sync,
            util::bind_front(&performance_counter::extract_value<T>));
    }
template <typename T>
T get_value(launch::sync_policy, error_code& ec = throws) const
{
    return get_counter_value(launch::sync).get_value<T>(ec);
}
};

// Return all counters matching the given name (with optional wild cards).
HPX_EXPORT std::vector<performance_counter> discover_counters(
    std::string const& name, error_code& ec = throws);
} } // namespace hpx::performance_counters

```

In order to implement a full performance counter, you have to create an *HPX* component exposing this interface. To simplify this task, *HPX* provides a ready-made base class which handles all the boiler plate of creating a component for you. The remainder of this section will explain the process of creating a full performance counter based on the Sine example, which you can find in the directory examples/performance_counters/sine/.

The base class is defined in the header file `base_performance_counter.cpp` as:

```

// Copyright (c) 2007-2018 Hartmut Kaiser
//
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#include <hpx/actions_base/component_action.hpp>
#include <hpx/components_base/component_type.hpp>
#include <hpx/components_base/server/component_base.hpp>
#include <hpx/performance_counters/counters.hpp>
#include <hpx/performance_counters/server/base_performance_counter.hpp>

///////////////////////////////
// [performance_counter_base_class
namespace hpx { namespace performance_counters {
    template <typename Derived>
    class base_performance_counter;
}} // namespace hpx::performance_counters
//]

/////////////////////////////
namespace hpx { namespace performance_counters {
    template <typename Derived>
    class base_performance_counter
        : public hpx::performance_counters::server::base_performance_counter
        , public hpx::components::component_base<Derived>
    {
        private:
            typedef hpx::components::component_base<Derived> base_type;
    public:

```

(continues on next page)

(continued from previous page)

```

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

base_performance_counter() = default;

base_performance_counter(
    hpx::performance_counters::counter_info const& info)
: base_type_holder(info)
{
}

// Disambiguate finalize() which is implemented in both base classes
void finalize()
{
    base_type_holder::finalize();
    base_type::finalize();
}

hpx::naming::address get_current_address() const
{
    return hpx::naming::address(
        hpx::naming::get_gid_from_locality_id(hpx::get_locality_id()),
        hpx::components::get_component_type<Derived>(),
        const_cast<Derived*>(static_cast<Derived const*>(this)));
}
};

} } // namespace hpx::performance_counters

```

The single template parameter is expected to receive the type of the derived class implementing the performance counter. In the Sine example this looks like:

```

// Copyright (c) 2007-2012 Hartmut Kaiser
//
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/hpx.hpp>
#include <hpx/include/lcos_local.hpp>
#include <hpx/include/performance_counters.hpp>
#include <hpx/include/util.hpp>

#include <cstdint>

namespace performance_counters { namespace sine { namespace server
{
    /////////////////////////////////
    //[sine_counter_definition
    class sine_counter
        : public hpx::performance_counters::base_performance_counter<sine_counter>
    []
}}
```

(continues on next page)

(continued from previous page)

```

{
public:
    sine_counter() : current_value_(0), evaluated_at_(0) {}
    explicit sine_counter(
        hpx::performance_counters::counter_info const& info);

    /// This function will be called in order to query the current value of
    /// this performance counter
    hpx::performance_counters::counter_value get_counter_value(bool reset);

    /// The functions below will be called to start and stop collecting
    /// counter values from this counter.
    bool start();
    bool stop();

    /// finalize() will be called just before the instance gets destructed
    void finalize();

protected:
    bool evaluate();

private:
    typedef hpx::lcos::local::spinlock mutex_type;

    mutable mutex_type mtx_;
    double current_value_;
    std::uint64_t evaluated_at_;

    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 `simplest_performance_counter.cpp` for the full source code of the counter definition). For more information about this technique (called Curiously Recurring Template Pattern - CRTP), please see for instance the corresponding [Wikipedia article¹³²](#). 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.

¹³² http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Existing HPX performance counters

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

Table 2.27: AGAS performance counters

Counter type	Counter instance formatting	Description	Parameters	
/agas/count/<agas_service>	<p>where:</p> <p><agas_service> is one of the following:</p> <ul style="list-style-type: none"> <i>primary namespace services</i>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate, begin_migration, end_migration <i>component namespace services</i>: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type <i>locality namespace services</i>: free, localities, num_localities, num_threads, resolve_locality, resolved_localities <i>symbol namespace services</i>: bind, resolve, unbind, iterate_names, on_symbol_namespace_event 	<p><agas_instance>/total where:</p> <p><agas_instance> is the name of the AGAS service to query. Currently, this value will be locality#0 where 0 is the root <i>locality</i> (the id of the locality hosting the AGAS service).</p> <p>The value for * can be any <i>locality</i> id for the following <agas_service>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, bind, resolve, unbind, and iterate_names (only the primary and symbol AGAS service components live on all localities, whereas all other AGAS services are available on locality#0 only).</p>	None	Returns the total number of invocations of the specified AGAS service since its creation.
/agas/<agas_service_category>/count	<p>where:</p> <p><agas_service_category> is one of the following: primary, locality, component or symbol</p>	<p><agas_instance>/total where:</p> <p><agas_instance> is the name of the AGAS service to query. Currently, this value will be locality#0 where 0 is the root <i>locality</i> (the id of the locality hosting the AGAS service). Except for <agas_service_category>, primary or symbol for which the value for * can be any <i>locality</i> id (only the primary and symbol AGAS service components live on all localities, whereas all other AGAS services are available on locality#0 only).</p>	None	Returns the overall total number of invocations of all AGAS services provided by the given AGAS service category since its creation.
agas/time/<agas_service>	<p>where:</p> <p><agas_service> is one of the following:</p> <ul style="list-style-type: none"> <i>primary namespace services</i>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate, begin_migration, end_migration <i>component namespace services</i>: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type <i>locality namespace services</i>: free, localities, num_localities, num_threads, resolve_locality, resolved_localities <i>symbol namespace services</i>: bind, resolve, unbind, iterate_names, on_symbol_namespace_event 	<p><agas_instance>/total where:</p> <p><agas_instance> is the name of the AGAS service to query. Currently, this value will be locality#0 where 0 is the root <i>locality</i> (the id of the locality hosting the AGAS service).</p> <p>The value for * can be any <i>locality</i> id for the following <agas_service>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, bind, resolve, unbind, and iterate_names (only the primary and symbol AGAS service components live on all localities, whereas all other AGAS services are available on locality#0 only).</p>	None	Returns the overall execution time of the specified AGAS service since its creation (in nanoseconds).
164 locality namespace services: free, localities, num_localities, num_threads, resolve_locality, resolved_localities	Chapter 2: What's so special about HPX?			

Table 2.28: 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. <code>en</code> or <code>eceived</code>) for the specified <connection_type>. The performance counters for the connection type <code>mpi</code> are available only if the compile time constant <code>HPX_HAVE_PARCELPORT_MPI</code> was defined while compiling the <i>HPX</i> core library (which is not defined by default, the corresponding cmake configuration constant is <code>HPX_WITH_PARCELPORT_MPI</code>). 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. <code>en</code> or <code>eceived</code>). The performance counters for the connection type <code>mpi</code> are available only if the compile time constant <code>HPX_HAVE_PARCELPORT_MPI</code> was defined while compiling the <i>HPX</i> core library (which is not defined by default, the corresponding cmake configuration constant is <code>HPX_WITH_PARCELPORT_MPI</code>). 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. <code>sent</code> or <code>received</code> possibly compressed) for the specified <connection_type> by the given <i>locality</i> . The performance counters for the connection type <code>mpi</code> are available only if the compile time constant <code>HPX_HAVE_PARCELPORT_MPI</code> was defined while compiling the <i>HPX</i> core library (which is not defined by default, the corresponding cmake configuration constant is <code>HPX_WITH_PARCELPORT_MPI</code>). Please see <i>CMake variables used to configure HPX</i> for more details.	If the config-time option <code>-DHPX_WITH_PARCELPORT_ACTIONS</code> was specified, this counter allows one to specify an optional action name as its parameter. In this case the counter will report the number of bytes transmitted for the given action only.
/serialize/ time/ <connection_type> <operation>	locality#*/ total where: * is the <i>locality</i> id of the <i>locality</i> .	Returns the overall time spent performing outgoing data serialization for the specified <connection_type> on the given <i>locality</i> (see <operation>, e.g. <code>sent</code> or <code>received</code>). The performance counters for the connection	If the config-time option <code>-DHPX_WITH_PARCELPORT_ACTIONS</code> was specified, this counter allows one

Table 2.29: Thread manager performance counters

Counter type	Counter instance format- ting	Description	Parameters
/threads/count/ cumulative	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#* where: locality#* is defining the <i>locality</i> for which the overall number of retired HPX-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 HPX-threads should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the 'default' pool.</p>	Returns the overall number of executed (retired) HPX-threads on the given <i>locality</i> since application start. If the instance name is <i>total</i> the counter returns the accumulated number of retired HPX-threads for all worker threads (cores) on that <i>locality</i> . If the instance name is <i>worker-thread#*</i> the counter will return the overall number of retired HPX-threads for all worker threads separately which the current value of the available only if the configuration time constant <code>HPX_WITH_THREAD_CUMULATIVE_COUNTS</code>	None

continues on next page

¹³³ A message can potentially consist of more than one *parcel*.

Table 2.29 – continued from previous page

/threads/time/ average	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the average time spent executing one <i>HPX</i>-thread 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 average time spent executing one <i>HPX</i>-thread should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</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 <i>total</i> the counter 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 <i>worker-thread#*</i> the counter will return the average time spent executing one <i>HPX</i>-thread for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_WITH_THREAD_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].</p>	None
---------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ average-overhead	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the average overhead spent executing one <i>HPX</i>-thread 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 average overhead spent executing one <i>HPX</i>-thread should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	<p>Returns the average time spent on overhead while executing one <i>HPX</i>-thread on the given <i>locality</i> since application start. If the instance name is total the counter returns the average time spent on overhead while executing one <i>HPX</i>-thread for all worker threads (cores) on that <i>locality</i>. If the instance name is worker-thread#* the counter will return the average time spent on overhead executing one <i>HPX</i>-thread for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_WITH_THREAD_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].</p>	None
------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/ cumulative-phases	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the overall number of executed HPX-thread phases (invocations) 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 executed HPX-thread phases (invocations) should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <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 HPX-thread phases (invocations) on the given <i>locality</i> since application start. If the instance name is <i>total</i> the counter returns the accumulated number of executed HPX-thread phases (invocations) for all worker threads (cores) on that <i>locality</i>. If the instance name is <i>worker-thread#*</i> the counter will return the overall number of executed HPX-thread phases for all worker threads separately. This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_CUMULATIVE_COUNTS</code> is set to ON (default: ON). The unit of measure for this counter is nanosecond [ns].</p>	None
--------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ average-phase	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the average time spent executing one <i>HPX</i>-thread phase (invocation) 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 average time executing one <i>HPX</i>-thread phase (invocation) should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	<p>Returns the average time spent executing one <i>HPX</i>-thread phase (invocation) on the given <i>locality</i> since application start. If the instance name is <i>total</i> the counter returns the average time spent executing one <i>HPX</i>-thread phase (invocation) for all worker threads (cores) on that <i>locality</i>. If the instance name is <i>worker-thread#*</i> the counter will return the average time spent executing one <i>HPX</i>-thread phase for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_WITH_THREAD_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].</p>	None
---------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ average-phase-overhead	locality#*/total locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#* where: locality#* is defining the <i>locality</i> for which the average time overhead executing one <i>HPX</i> -thread phase (invocation) should be queried for. The <i>locality</i> id (given by *) is a (zero based) number identifying the <i>locality</i> . pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the average overhead executing one <i>HPX</i> -thread phase (invocation) should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code> . If no pool-name is specified the counter refers to the ‘default’ pool.	Returns the average time spent on overhead executing one <i>HPX</i> -thread phase (invocation) on the given <i>locality</i> since application start. If the instance name is <i>total</i> the counter returns the average time spent on overhead while executing one <i>HPX</i> -thread phase (invocation) for all worker threads (cores) on that <i>locality</i> . If the instance name is <i>worker-thread#*</i> the counter will return the average time spent on overhead executing one <i>HPX</i> -thread phase for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_WITH_THREAD_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].	None
------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ overall	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the overall time spent running the scheduler 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 time spent running the scheduler should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	<p>Returns the overall time spent running the scheduler on the given <i>locality</i> since application start. If the instance name is <i>total</i> the counter returns the overall time spent running the scheduler for all worker threads (cores) on that <i>locality</i>. If the instance name is <i>worker-thread#*</i> the counter will return the overall time spent running the scheduler for all worker threads separately. This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_IDLE_RATES</code> is set to <code>ON</code> (default: <code>OFF</code>). The unit of measure for this counter is nanosecond [ns].</p>	None
---------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ cumulative	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the overall time spent executing all <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 time spent executing all <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 time spent executing all <i>HPX</i>-threads on the given <i>locality</i> since application start. If the instance name is <i>total</i> the counter returns the overall time spent executing all <i>HPX</i>-threads for all worker threads (cores) on that <i>locality</i>. If the instance name is <i>worker-thread#*</i> the counter will return the overall time spent executing all <i>HPX</i>-threads for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_THREAD_MAINTAIN_IDLE_RATES</code> are set to ON (default: OFF).</p>	None
------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ cumulative-overhead	sor locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#* where: locality#* is defining the <i>locality</i> for which the overall overhead time in- curred by executing all <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> . pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the the over- all overhead time incurred by executing all <i>HPX</i> - threads should be queried for. The worker thread number (given by the * is a (zero based) num- ber identifying the worker thread. The number of available worker threads is usually specified on the command line for the ap- plication using the option --hpx:threads. If no pool-name is specified the counter refers to the ‘de- fault’ pool.	Returns the overall overhead time incurred executing all <i>HPX</i> -threads on the given <i>locality</i> since application start. If the instance name is total the counter returns the overall overhead time incurred executing all <i>HPX</i> -threads for all worker threads (cores) on that <i>locality</i> . If the instance name is worker-thread#* the counter will return the overall overhead time incurred executing all <i>HPX</i> -threads for all worker threads sepa- rately. This counter is available only if the con- figuration time constants HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS (default: ON) and HPX_THREAD_MAINTAIN_IDLE_RATES are set to ON (default: OFF). The unit of mea- sure for this counter is nanosecond [ns].	None
---------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

threads/count/ instantaneous/ <thread-state> where: <thread-state> is one of the following: all, active, pending, suspended, terminated, staged	locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#* where: locality#* is defining the <i>locality</i> for which the current number of threads with the given state should be queried for. The <i>locality</i> id (given by * is a (zero based) number identifying the <i>locality</i> . pool#* is defining the pool for which the cur- rent value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current number of threads with the given state should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool. The staged thread state refers to registered tasks before they are converted to thread objects.	Returns the current number of HPX-threads having the given thread state on the given <i>locality</i> . If the instance name is total the counter returns the current num- ber of HPX-threads of the given state for all worker threads (cores) on that <i>locality</i> . If the instance name is worker-thread#* the counter will return the current number of HPX- threads in the given state for all worker threads separately.	None
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

<p>threads/ wait-time/ <thread-state></p> <p>where: <thread-state> is one of the following: pending staged</p>	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the average wait time of <i>HPX</i>-threads (pending) or thread descriptions (staged) with the given state 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 average wait time for the given state should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p> <p>The staged thread state refers to the wait time of registered tasks before they are converted into thread objects, while the pending thread state refers to the wait time of threads in any of the scheduling queues.</p>	<p>Returns the average wait time of <i>HPX</i>-threads (if the thread state is pending or of task descriptions (if the thread state is staged on the given <i>locality</i> since application start. If the instance name is total the counter returns the wait time of <i>HPX</i>-threads of the given state for all worker threads (cores) on that <i>locality</i>. If the instance name is worker-thread#* the counter will return the wait time of <i>HPX</i>-threads in the given state for all worker threads separately. These counters are available only if the compile time constant <code>HPX_WITH_THREAD_QUEUE_WAITTIME</code> was defined while compiling the <i>HPX</i> core library (default: OFF). The unit of measure for this counter is nanosecond [ns].</p>	<p>None</p>
--------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------

continues on next page

Table 2.29 – continued from previous page

/threads/ idle-rate	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the average idle rate of all (or one) worker 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 averaged idle rate should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the average idle rate for the given worker thread(s) on the given <i>locality</i> . The idle rate is defined as the ratio of the time spent on scheduling and management tasks and the overall time spent executing work since the application started. This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_IDLE_RATES</code> is set to ON (default: OFF).	None
------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/ creation-idle-rate	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the average creation idle rate of all (or one) worker 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 averaged idle rate should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	<p>Returns the average idle rate for the given worker thread(s) on the given <i>locality</i> which is caused by creating new threads. The creation idle rate is defined as the ratio of the time spent on creating new threads and the overall time spent executing work since the application started. This counter is available only if the configuration time constants <code>HPX_WITH_THREAD_IDLE_RATES</code> (default: OFF) and <code>HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES</code> are set to ON.</p>	None
---------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/ cleanup-idle-rate	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the average cleanup idle rate of all (or one) worker 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 averaged cleanup idle rate should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	<p>Returns the average idle rate for the given worker thread(s) on the given <i>locality</i> which is caused by cleaning up terminated threads. The cleanup idle rate is defined as the ratio of the time spent on cleaning up terminated thread objects and the overall time spent executing work since the application started. This counter is available only if the configuration time constants <code>HPX_WITH_THREAD_IDLE_RATES</code> (default: OFF) and <code>HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES</code> are set to ON.</p>	None
--------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threadqueue/ length	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the current length of all thread queues in the scheduler for all (or one) worker 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 current length of all thread queues in the scheduler should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the overall length of all queues for the given worker thread(s) on the given <i>locality</i> .	None
/threads/count/ stack-unbinds	<p>locality#*/total where:</p> <p>* is the <i>locality</i> id of the <i>locality</i> the unbind (madvise) operations should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i>.</p>	Returns the total number of HPX-thread unbind (madvise) operations performed for the referenced <i>locality</i> . Note that this counter is not available on Windows based platforms.	None

continues on next page

Table 2.29 – continued from previous page

/threads/count/ stack-recycles	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the recycling operations should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the total number of <i>HPX</i> -thread recycling operations performed.	None
/threads/count/ stolen-from-pending	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the number of ‘stole’ threads should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the total number of <i>HPX</i> -threads ‘stolen’ from the pending thread queue by a neighboring thread worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).	None

continues on next page

Table 2.29 – continued from previous page

/threads/count/pending-misses	<p>locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the number of pending queue misses of all (or one) worker 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 number of pending queue misses should be queried for. The worker thread number (given by *) 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>	Returns the total number of times that the referenced worker-thread on the referenced <i>locality</i> failed to find pending HPX-threads in its associated queue. This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).	None
-------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/pending-accesses	<p>locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the number of pending queue accesses of all (or one) worker 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 number of pending queue accesses should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the total number of times that the referenced worker-thread on the referenced <i>locality</i> looked for pending HPX-threads in its associated queue. This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).	None
---------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/ stolen-from-staged	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the number of <i>HPX</i>-threads stolen from the staged queue of all (or one) worker 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 number of <i>HPX</i>-threads stolen from the staged queue should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the total number of <i>HPX</i> -threads ‘stolen’ from the staged thread queue by a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).	None
---------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/ stolen-to-pending	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the number of <i>HPX</i>-threads stolen to the pending queue of all (or one) worker 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 number of <i>HPX</i>-threads stolen to the pending queue should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the total number of <i>HPX</i> -threads ‘stolen’ to the pending thread queue of the worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).	None
--------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/ stolen-to-staged	<p>locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the number of <i>HPX</i>-threads stolen to the staged queue of all (or one) worker 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 number of <i>HPX</i>-threads stolen to the staged queue should be queried for. The worker thread number (given by the * is a (zero based) worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the total number of <i>HPX</i> -threads ‘stolen’ to the staged thread queue of a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).	None
-------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/ objects	locality#*/total or locality#*/ allocator#* where: locality#* is defining the <i>locality</i> for which the current (cumulative) num- ber of all created <i>HPX</i> - thread objects should be queried for. The <i>locality</i> id (given by * is a (zero based) number identifying the <i>locality</i> . allocator#* is defin- ing the number of the allo- cator instance using which the threads have been cre- ated. <i>HPX</i> uses a vary- ing number of allocators to create (and recycle) <i>HPX</i> -thread objects, most likely these counters are of use for debugging pur- poses only. The allocator id (given by * is a (zero based) number identifying the allocator to query.	Returns the total num- ber of <i>HPX</i> -thread ob- jects created. Note that thread objects are reused to improve system perfor- mance, thus this number does not reflect the num- ber of actually executed (retired) <i>HPX</i> -threads.	None
/scheduler/ utilization/ instantaneous	locality#*/total where: locality#* is defining the <i>locality</i> for which the current (instantaneous) scheduler utilization queried for. The <i>locality</i> id (given by * is a (zero based) number identifying the <i>locality</i> .	Returns the total (instantaneous) scheduler utilization. This is the current percentage of scheduler threads executing <i>HPX</i> threads.	Percent

continues on next page

Table 2.29 – continued from previous page

/threads/ idle-loop-count/ instantaneous	locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#* where: locality#* is defining the <i>locality</i> for which the current accumulated value of all idle-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> . pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current value of the idle-loop counter should be queried for. The worker thread number (given by the * is a (zero based) worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code> . If no pool-name is specified the counter refers to the ‘default’ pool.	Returns the current (instantaneous) idle-loop count for the given HPX-worker thread or the accumulated value for all worker threads.	None
------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/ busy-loop-count/ instantaneous	<p>locality#*/ worker-thread#*</p> <p>or</p> <p>locality#*/ pool#*/ worker-thread#*</p> <p>where:</p> <p>locality#* is defining the <i>locality</i> for which the 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>.</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 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>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the current (instantaneous) busy-loop count for the given HPX-worker thread or the accumulated value for all worker threads.	None
------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/background-work-duration	locality#*/total locality#*/ worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads.	Returns the overall time spent performing background work on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].	None
----------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/ background-overhead	locality#*/total or locality#*/ worker-thread#* where: locality#* is defining the locality for which the background overhead should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the background overhead should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads.	Returns the background overhead on the given locality since application start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%.	None
----------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/background-send-duration	<p>locality#*/total locality#*/worker-thread#*</p> <p>where:</p> <p>locality#* is defining the locality for which the overall time spent performing background work related to sending parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality.</p> <p>worker-thread#* is defining the worker thread for which the overall time spent performing background work related to sending parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>.</p>	<p>Returns the overall time spent performing background work related to sending parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_WITH_BACKGROUND_THREAD_COUNTERS</code> (default: OFF) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcel-port only.</p>	None
----------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/ background-send-overhead	locality#*/total locality#*/ worker-thread#* where: locality#* is defining the locality for which the background overhead related to sending parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the background overhead related to sending parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads.	Returns the background overhead related to sending parcels on the given locality since application start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcel-port only.	None
---------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ background-receive	locality#*/total duration locality#*/ worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work related to receiving parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent per- forming background work related to receiving parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the app- lication using the option --hpx:threads.	Returns the overall time spent performing back- ground work related to receiving parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent per- forming background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent per- forming background work for all worker threads separately. This counter is available only if the con- figuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of mea- sure for this counter is nanosecond [ns]. This counter will cur- rently return meaningful values for the MPI parcel- port only.	None
--------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

continues on next page

Table 2.29 – continued from previous page

/threads/ background-receive	locality#*/total overhead locality#*/ worker-thread#* where: locality#* is defining the locality for which the background overhead re- lated to receiving should be queried for. The lo- cality id (given by *) is a (zero based) number iden- tifying the locality. worker-thread#* is defining the worker thread for which the background overhead related to receiving parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the ap- plication using the option --hpx:threads.	Returns the background overhead related to re- ceiving parcels on the given locality since ap- plication start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configura- tion time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of mea- sure displayed for this counter is 0.1%. This counter will cur- rently return meaningful values for the MPI parcel- port only.	None
---------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------

Table 2.30: 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
196		Chapter 2. What's so special about HPX?	
/runtime/memory/total	locality#*/total where:	Returns the total available memory for use by the referenced locality (in bytes)	None

Table 2.31: Performance counters exposing PAPI hardware counters

Counter type	Counter instance formatting	Description	Pa-ram-e-ters
/papi/<papi_event> where: <papi_event> is the name of the PAPI event to expose as a performance counter (such as PAPI_SR_INS). Note that the list of available PAPI events changes depending on the used architecture. For a full list of available PAPI events and their (short) description use the --hpx:list-counters and --hpx:papi-event-info= command line options.	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the <i>locality</i> for which the 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 aliasing the option --hpx:threads.	This counter returns the current count of occurrences of the specified PAPI event. This counter is available only if the configuration time constant HPX_WITH_PAPI is set to ON (default: OFF).	None

Table 2.32: Performance counters for general statistics

Counter type	Counter instance formatting	Description	Parameters
/	Any full performance counter average. 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.
/	Any full performance counter rolling average. 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.
/	Any full performance counter standard deviation. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current standard deviation (stddev) value calculated based on the values queried from the underlying counter (the one specified as the instance name).	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume 1000 [ms] as the default. The second value can be either 0 or 1 and specifies whether the underlying counter should be reset during evaluation 1 or not 0. The default value is 0.
/	Any full performance counter rolling standard deviation. 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.
/	Any full performance counter median. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current (statistically estimated) median value calculated based on the values queried from the underlying counter (the one specified as the instance name).	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume 1000 [ms] as the default. The second value can be either 0 or 1 and specifies whether the underlying counter should be reset during evaluation 1 or not 0. The default value is 0.
198			Chapter 2. What's so special about HPX?

Table 2.33: Performance counters for elementary arithmetic operations

Counter type	Counter instance formatting	Description	Parameters
/arithmetics/add	None	Returns the sum calculated based on the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
/arithmetics/subtract	None	Returns the difference calculated based on the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
/arithmetics/multiply	None	Returns the product calculated based on the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
/arithmetics/divide	None	Returns the result of division of the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
/arithmetics/mean	None	Returns the average value of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
/arithmetics/variance	None	Returns the standard deviation of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
2.3. Manual			199
/arithmetics/median	None	Returns the median value of all values queried from	The parameter will be interpreted as a comma sep-

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

```
$ ./bin/hello_world_distributed -t2 \
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
    --hpx:print-counter=/arithmetics/add@/threads{locality#0/worker-thread#*}/count/
    ↪cumulative
hello world from OS-thread 0 on locality 0
hello world from OS-thread 1 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.515640,[s],25
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.515520,[s],36
/arithmetics/add@/threads{locality#0/worker-thread#*}/count/cumulative,1,0.516445,[s],
    ↪64
```

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

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

Table 2.34: Performance counters tracking parcel coalescing

Counter type	Counter instance formatting	Description	Parameters
/coalesced/ count	where: parcel 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> .
/coalesced/ count	where: message 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> .
/coalesced/ count	where: average 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>
2.3. Manual	/coalesced/ time	where: average is the arrival	Returns the average time between arriving parcels for the action which is given by the counter parameter.

Note: The performance counters related to *parcel* coalescing are available only if the configuration time constant `HPX_WITH_PARCEL_COALESCING` is set to ON (default: ON). However, even in this case it will be available only for actions that are enabled for parcel coalescing (see the macros `HPX_ACTIONUSESMESSAGECOALESCING` and `HPX_ACTIONUSESMESSAGECOALESCING_NOTHROW`).

APEX integration

HPX provides integration with [APEX](#)¹³⁴, which is a framework for application profiling using task timers and various performance counters Huck *et al.*¹³⁷. It can be added as a git submodule by turning on the option `HPX_WITH_APEX:BOOL` during CMake configuration. [TAU](#)¹³⁵ is an optional dependency when using APEX.

To build *HPX* with APEX, add `HPX_WITH_APEX=ON`, and, optionally, `TAU_ROOT=$PATH_TO_TAU` to your CMake configuration. In addition, you can override the tag used for APEX with the `HPX_WITH_APEX_TAG` option. Please see the [APEX HPX documentation](#)¹³⁶ for detailed instructions on using APEX with *HPX*.

References

2.3.10 *HPX* runtime and resources

HPX thread scheduling policies

The *HPX* runtime has five thread scheduling policies: local-priority, static-priority, local, static and abp-priority. These policies can be specified from the command line using the command line option `--hpx:queuing`. In order to use a particular scheduling policy, the runtime system must be built with the appropriate scheduler flag turned on (e.g. `cmake -DHPX_THREAD_SCHEDULERS=local`, see *CMake variables used to configure HPX* for more information).

Priority local scheduling policy (default policy)

- default or invoke using: `--hpx: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.

¹³⁴ <http://uo-oaciss.github.io/apex>

¹³⁷ K. A. Huck, A. Porterfield, N Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler. *An autonomic performance environment for exascale*. Supercomputing Frontiers and Innovations, 2015.

¹³⁵ <https://www.cs.uoregon.edu/research/tau/home.php>

¹³⁶ <https://uo-oaciss.github.io/apex/usage/#hpx-louisiana-state-university>

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

Static priority scheduling policy

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

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

Local scheduling policy

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

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

Static scheduling policy

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

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

Priority ABP scheduling policy

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

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

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

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 [MPI¹³⁸](#) 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 [MPI¹³⁹](#) communication. [MPI¹⁴⁰](#) tasks will then always run on the same thread, instead of potentially being stuck in a queue behind other threads.

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

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

Using the resource partitioner

The `hpx::resource::partitioner` is now created during *HPX* runtime initialization without explicit action needed from the user. To specify some of the initialization parameters you can use the `hpx::init_params`.

The resource partitioner callback is the interface to add thread pools to the *HPX* runtime and to assign resources to the thread pools. In order to create custom thread pools you can specify the resource partitioner callback `hpx::init_params::rp_callback` which will be called once the resource partitioner will be created , see the example below. You can also specify other parameters, see `hpx::init_params`.

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

Then, to add resources to the thread pool you can use the `hpx::resource::partitioner::add_resource` method. The resource partitioner exposes the hardware topology retrieved using [Portable Hardware Locality \(HWLOC\)](#)¹⁴¹ and lets you iterate through the topology to add the wanted processing units to the thread pool. Below is an example of adding all processing units from the first NUMA domain to a custom thread pool, unless there is only one NUMA domain in which case we leave the first processing unit for the default thread pool:

Note: Whatever processing units not assigned to a thread pool by the time `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.

¹³⁸ https://en.wikipedia.org/wiki/Message_Passing_Interface

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

¹⁴⁰ https://en.wikipedia.org/wiki/Message_Passing_Interface

¹⁴¹ <https://www.open-mpi.org/projects/hwloc/>

Difference between the old and new version

In the old version, you had to create an instance of the `resource_partitioner` with `argc` and `argv`.

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

From *HPX 1.5.0* onwards, you just pass `argc` and `argv` to `hpx::init()` or `hpx::start()` for the binding options to be parsed by the resource partitioner.

```
int main(int argc, char** argv)
{
    hpx::init_params init_args;
    hpx::init(argc, argv, init_args);
}
```

In the old version, when creating a custom thread pool, you just called the utilities on the resource partitioner instantiated previously.

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

    rp.create_thread_pool("my-thread-pool");

    bool one numa_domain = rp.numa_domains().size() == 1;
    bool skipped_first_pu = false;

    hpx::resource::numa_domain const& d = rp.numa_domains()[0];

    for (const hpx::resource::core& c : d.cores())
    {
        for (const hpx::resource::pu& p : c.pus())
        {
            if (one numa_domain && !skipped_first_pu)
            {
                skipped_first_pu = true;
                continue;
            }

            rp.add_resource(p, "my-thread-pool");
        }
    }

    hpx::init();
}
```

You now specify the resource partitioner callback which will tie the resources to the resource partitioner created during runtime initialization.

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

(continues on next page)

(continued from previous page)

```

bool one numa_domain = rp.numa_domains().size() == 1;
bool skipped_first_pu = false;

hpx::resource::numa_domain const& d = rp.numa_domains()[0];

for (const hpx::resource::core& c : d.cores())
{
    for (const hpx::resource::pu& p : c.pus())
    {
        if (one numa_domain && !skipped_first_pu)
        {
            skipped_first_pu = true;
            continue;
        }

        rp.add_resource(p, "my-thread-pool");
    }
}

int main(int argc, char* argv[])
{
    hpx::init_params init_args;
    init_args.rp_callback = &init_resource_partitioner_handler;

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

```

Advanced usage

It is possible to customize the built in schedulers by passing scheduler options to `hpx::resource::partitioner::create_thread_pool`. It is also possible to create and use custom schedulers.

Note: It is not recommended to create your own scheduler. The HPX developers use this to experiment with new scheduler designs before making them available to users via the standard mechanisms of choosing a scheduler (command line options). If you would like to experiment with a custom scheduler the resource partitioner example `shared_priority_queue_scheduler.cpp` contains a fully implemented scheduler with logging etc. to make exploration easier.

To choose a scheduler and custom mode for a thread pool, pass additional options when creating the thread pool like this:

```

rp.create_thread_pool("my-thread-pool",
    hpx::resource::policies::local_priority_lifo,
    hpx::policies::scheduler_mode(
        hpx::policies::scheduler_mode::default_ |
        hpx::policies::scheduler_mode::enable_elasticity));

```

The available schedulers are documented here: `hpx::resource::scheduling_policy`, and the available scheduler modes here: `hpx::threads::policies::scheduler_mode`. Also see the examples folder for examples of advanced resource partitioner usage: `simple_resource_partitioner.cpp` and `oversubscribing_resource_partitioner.cpp`.

2.3.11 Miscellaneous

Error handling

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

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

Working with exceptions

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

```
#include <hpx/iostream.hpp>
#include <hpx/modules/runtime_local.hpp>

// [[error_handling_raise_exception
void raise_exception()
```

The exception is thrown using the macro `HPX_THROW_EXCEPTION`. The type of the thrown exception is `hpx::exception`. This associates additional diagnostic information with the exception, such as file name and line number, *locality* id and thread id, and stack backtrace from the point where the exception was thrown.

Any exception thrown during the execution of an action is transferred back to the (asynchronous) invocation site. It will be rethrown in this context when the calling thread tries to wait for the result of the action by invoking either `future<>::get()` or the synchronous action invocation wrapper as shown here:

```
{
    ///////////////////////////////////////////////////////////////////
    // Error reporting using exceptions
    // [[exception_diagnostic_information
    hpx::cout << "Error reporting using exceptions\n";
    try {
        // invoke raise_exception() which throws an exception
        raise_exception_action do_it;
        do_it(hpx::find_here());
    }
    catch (hpx::exception const& e) {
        // Print just the essential error information.
        hpx::cout << "caught exception: " << e.what() << "\n\n";
        // Print all of the available diagnostic information as stored with
        // the exception.
```

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

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

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

```
//]

// Detailed error reporting using exceptions
//[exception_diagnostic_elements
hpx::cout << "Detailed error reporting using exceptions\n";
try {
    // Invoke raise_exception() which throws an exception.
    raise_exception_action do_it;
    do_it(hpx::find_here());
}
catch (hpx::exception const& e) {
    // Print the elements of the diagnostic information separately.
    hpx::cout << "{what}: "           << hpx::get_error_what(e) << "\n";
    hpx::cout << "{locality-id}: "   << hpx::get_error_locality_id(e) << "\n";
    hpx::cout << "{hostname}: "      << hpx::get_error_host_name(e) << "\n";
    hpx::cout << "{pid}: "          << hpx::get_error_process_id(e) << "\n";
    hpx::cout << "{function}: "     << hpx::get_error_function_name(e) << "\n";
    hpx::cout << "{file}: "         << hpx::get_error_file_name(e) << "\n";
    hpx::cout << "{line}: "         << hpx::get_error_line_number(e) << "\n";
    hpx::cout << "{os-thread}: "    << hpx::get_error_os_thread(e) << "\n";
    hpx::cout << "{thread-id}: "   << std::hex << hpx::get_error_thread_id(e)
    << "\n";
    hpx::cout << "{thread-description}: "
    << hpx::get_error_thread_description(e) << "\n";
    hpx::cout << "{state}: "        << std::hex << hpx::get_error_state(e)
    << "\n";
}
```

Working with error codes

Most of the API functions exposed by *HPX* can be invoked in two different modes. By default those will throw an exception on error as described above. However, sometimes it is desirable not to throw an exception in case of an error condition. In this case an object instance of the *hpx::error_code* type can be passed as the last argument to the API function. In case of an error, the error condition will be returned in that *hpx::error_code* instance. The following example demonstrates extracting the full diagnostic information without exception handling:

```
///////////////////////////////
// Error reporting using error code
{
    //#[error_handling_diagnostic_information
    hpx::cout << "Error reporting using error code\n";

    // Create a new error_code instance.
    hpx::error_code ec;

    // If an instance of an error_code is passed as the last argument while
    // invoking the action, the function will not throw in case of an error
    // but store the error information in this error_code instance instead.
    raise_exception_action do_it;
    do_it(hpx::find_here(), ec);

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

(continues on next page)

(continued from previous page)

```
// Print all of the available diagnostic information as stored with
// the exception.
hpx::cout << "diagnostic information:"
```

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

This example shows how an error can be handled without having to resolve to exceptions and that the returned `hpx::error_code` instance can be used in a very similar way as the `hpx::exception` type above. Simply pass it to the `hpx::diagnostic_information`, which retrieves all available diagnostic information from the error code instance as a formatted string.

As for handling exceptions, when working with error codes, under certain circumstances it is desirable to output only some of the diagnostics, or to output those using different formatting. For this case, HPX exposes a set of lower-level functions usable with error codes as demonstrated in the following code snippet:

```
// Detailed error reporting using error code
{
    // [error_handling_diagnostic_elements
    hpx::cout << "Detailed error reporting using error code\n";

    // Create a new error_code instance.
    hpx::error_code ec;

    // If an instance of an error_code is passed as the last argument while
    // invoking the action, the function will not throw in case of an error
    // but store the error information in this error_code instance instead.
    raise_exception_action do_it;
    do_it(hpx::find_here(), ec);

    if (ec) {
        // Print the elements of the diagnostic information separately.
        hpx::cout << "{what}: "           << hpx::get_error_what(ec) << "\n";
        hpx::cout << "{locality-id}: "   << hpx::get_error_locality_id(ec) << "\n";
        hpx::cout << "{hostname}: "      << hpx::get_error_host_name(ec) << "\n";
        hpx::cout << "{pid}: "          << hpx::get_error_process_id(ec) << "\n";
        hpx::cout << "{function}: "     << hpx::get_error_function_name(ec)
            << "\n";
        hpx::cout << "{file}: "         << hpx::get_error_file_name(ec) << "\n";
        hpx::cout << "{line}: "         << hpx::get_error_line_number(ec) << "\n";
        hpx::cout << "{os-thread}: "    << hpx::get_error_os_thread(ec) << "\n";
        hpx::cout << "{thread-id}: "   << std::hex
            << hpx::get_error_thread_id(ec) << "\n";
        hpx::cout << "{thread-description}: "
            << hpx::get_error_thread_description(ec) << "\n\n";
        hpx::cout << "{state}: "        << std::hex << hpx::get_error_state(ec)
            << "\n";
        hpx::cout << "{stack-trace}: "  << hpx::get_error_backtrace(ec) << "\n";
    }
}
```

For more information please refer to the documentation of `hpx::get_error_what`, `hpx::get_error_locality_id`, `hpx::get_error_host_name`, `hpx::get_error_process_id`, `hpx::get_error_function_name`, `hpx::get_error_file_name`, `hpx::get_error_line_number`, `hpx::get_error_os_thread`, `hpx::get_error_thread_id`, `hpx::get_error_thread_description`, `hpx::get_error_backtrace`, `hpx::get_error_env`, and `hpx::get_error_state`.

Lightweight error codes

Sometimes it is not desirable to collect all the ambient information about the error at the point where it happened as this might impose too much overhead for simple scenarios. In this case, *HPX* provides a lightweight error code facility that will hold the error code only. The following snippet demonstrates its use:

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

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

    // If an instance of an error_code is passed as the last argument while
    // invoking the action, the function will not throw in case of an error
    // but store the error information in this error_code instance instead.
    raise_exception_action do_it;
    do_it(hpx::find_here(), ec);

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

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

All functions that retrieve other diagnostic elements from the `hpx::error_code` will fail if called with a lightweight error_code instance.

Utilities in *HPX*

In order to ease the burden of programming, *HPX* provides several utilities to users. The following section documents those facilities.

Checkpoint

See *checkpoint*.

The HPX I/O-streams component

The *HPX* I/O-streams subsystem extends the standard C++ output streams `std::cout` and `std::cerr` to work in the distributed setting of an *HPX* application. All of the output streamed to `hpx::cout` will be dispatched to `std::cout` on the console *locality*. Likewise, all output generated from `hpx::cerr` will be dispatched to `std::cerr` on the console *locality*.

Note: All existing standard manipulators can be used in conjunction with `hpx::cout` and `hpx::cerr`. Historically, *HPX* also defines `hpx::endl` and `hpx::flush` but those are just aliases for the corresponding standard manipulators.

In order to use either `hpx::cout` or `hpx::cerr`, application codes need to `#include <hpx/include/iostreams.hpp>`. For an example, please see the following ‘Hello world’ program:

```
// Copyright (c) 2007-2012 Hartmut Kaiser
//
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

///////////////////////////////
// The purpose of this example is to execute a HPX-thread printing
// "Hello World!" once. That's all.

//[hello_world_1_getting_started
// Including 'hpx/hpx_main.hpp' instead of the usual 'hpx/hpx_init.hpp' enables
// to use the plain C-main below as the direct main HPX entry point.
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

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

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

```
include(HPX_AddExecutable)

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

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

2.3.12 Troubleshooting

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

Undefined reference to `boost::program_options`

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

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

if you are not linking to a compatible version of `Boost.ProgramOptions`. We recommend that you use `hpx::program_options`, which is part of *HPX*, as a replacement for `boost::program_options` (see *program_options*). Until you have migrated to use `hpx::program_options` we recommend that you always build `Boost`¹⁴² libraries and *HPX* with the same compiler and C++ standard.

Undefined reference to `hpx::cout`

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

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

This usually happens if you are trying to use *HPX* iostreams functionality such as `hpx::cout` but are not linking against it. The iostreams functionality is not part of the core *HPX* library, and must be linked to explicitly. Typically this can be solved by adding `COMPONENT_DEPENDENCIES iostreams` to a call to `add_hpx_library/` `add_hpx_executable/hpx_setup_target` if using CMake. See *Creating HPX projects* for more details.

2.4 Terminology

This section gives definitions for some of the terms used throughout the *HPX* documentation and source code.

Locality A locality in *HPX* describes a synchronous domain of execution, or the domain of bounded upper response time. This normally is just a single node in a cluster or a NUMA domain in a SMP machine.

Active Global Address Space

AGAS *HPX* incorporates a global address space. Any executing thread can access any object within the domain of the parallel application with the caveat that it must have appropriate access privileges. The model does not assume that global addresses are cache coherent; all loads and stores will deal directly with the site of the target object. All global addresses within a Synchronous Domain are assumed to be cache coherent for those processor cores that incorporate transparent caches. The Active Global Address Space used by *HPX* differs from research **PGAS**¹⁴³ 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

¹⁴² <https://www.boost.org/>

¹⁴³ <https://www.pgas.org/>

parallelism within the context of a given process, as well as the coarse grained parallelism achieved through concurrency of multiple processes of an executing user job. *HPX* processes provide a hierarchical name space within the framework of the active global address space and support multiple means of internal state access from external sources.

Parcel The Parcel is a component in *HPX* that communicates data, invokes an action at a distance, and distributes flow-control through the migration of continuations. Parcels bridge the gap of asynchrony between synchronous domains while maintaining symmetry of semantics between local and global execution. Parcels enable message-driven computation and may be seen as a form of “active messages”. Other important forms of message-driven computation predating active messages include [dataflow tokens¹⁴⁴](#), the J-machine’s¹⁴⁵ 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*.

2.5 Why *HPX*?

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

¹⁴⁴ http://en.wikipedia.org/wiki/Dataflow_architecture

¹⁴⁵ <http://en.wikipedia.org/wiki/J%20Machine>

2.5.1 ParalleX—a new execution model for future architectures

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

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

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

2.5.2 What is HPX?

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

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

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

HPX is designed by the STE||AR¹⁴⁶ Group (Systems Technology, Emergent Parallelism, and Algorithm Research) at Louisiana State University (LSU)¹⁴⁷'s Center for Computation and Technology (CCT)¹⁴⁸ to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STE||AR¹⁴⁹ is a research group focusing on system software solutions and scientific application development for hybrid and many-core hardware architectures.

¹⁴⁶ <https://stellar-group.org>

¹⁴⁷ <https://www.lsu.edu>

¹⁴⁸ <https://www.cct.lsu.edu>

¹⁴⁹ <https://stellar-group.org>

For more information about the STE||AR¹⁵⁰ Group, see *People*.

2.5.3 What makes our systems slow?

Estimates say that we currently run our computers at well below 100% efficiency. The theoretical peak performance (usually measured in **FLOPS**¹⁵¹)—floating point operations per second) is much higher than any practical peak performance reached by any application. This is particularly true for highly parallel hardware. The more hardware parallelism we provide to an application, the better the application must scale in order to efficiently use all the resources of the machine. Roughly speaking, we distinguish two forms of scalability: strong scaling (see [Amdahl's Law](#)¹⁵²) and weak scaling (see [Gustafson's Law](#)¹⁵³). Strong scaling is defined as how the solution time varies with the number of processors for a fixed **total** problem size. It gives an estimate of how much faster we can solve a particular problem by throwing more resources at it. Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size **per processor**. In other words, it defines how much more data can we process by using more hardware resources.

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

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

Each of those four factors manifests itself in multiple and different ways; each of the hardware architectures and programming models expose specific forms. However, the interesting part is that all of them are limiting the scalability of applications no matter what part of the hardware jungle we look at. Hand-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, and it allows us to focus our analysis on very concrete patterns and measurable metrics. Moreover, any derived results will be applicable to a wide variety of targets.

2.5.4 Technology demands new response

Today's computer systems are designed based on the initial ideas of [John von Neumann](#)¹⁵⁴, as published back in 1945, and later extended by the [Harvard architecture](#)¹⁵⁵. These ideas form the foundation, the execution model, of computer systems we use currently. However, a new response is required in the light of the demands created by today's technology.

So, what are the overarching objectives for designing systems allowing for applications to scale as they should? In our opinion, the main objectives are:

¹⁵⁰ <https://stellar-group.org>

¹⁵¹ <http://en.wikipedia.org/wiki/FLOPS>

¹⁵² http://en.wikipedia.org/wiki/Amdahl%27s_law

¹⁵³ http://en.wikipedia.org/wiki/Gustafson%27s_law

¹⁵⁴ <http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf>

¹⁵⁵ http://en.wikipedia.org/wiki/Harvard_architecture

- Performance: as previously mentioned, scalability and efficiency are the main criteria people are interested in.
- Fault tolerance: the low expected mean time between failures ([MTBF¹⁵⁶](#)) of future systems requires embracing faults, not trying to avoid them.
- Power: minimizing energy consumption is a must as it is one of the major cost factors today, and will continue to rise in the future.
- Generality: any system should be usable for a broad set of use cases.
- Programmability: for programmer this is a very important objective, ensuring long term platform stability and portability.

What needs to be done to meet those objectives, to make applications scale better on tomorrow's architectures? Well, the answer is almost obvious: we need to devise a new execution model—a set of governing principles for the holistic design of future systems—targeted at minimizing the effect of the outlined **SLOW** factors. Everything we create for future systems, every design decision we make, every criteria we apply, have to be validated against this single, uniform metric. This includes changes in the hardware architecture we prevalently use today, and it certainly involves new ways of writing software, starting from the operating system, runtime system, compilers, and at the application level. However, the key point is that all those layers have to be co-designed; they are interdependent and cannot be seen as separate facets. The systems we have today have been evolving for over 50 years now. All layers function in a certain way, relying on the other layers to do so. But we do not have the time to wait another 50 years for a new coherent system to evolve. The new paradigms are needed now—therefore, co-design is the key.

2.5.5 Governing principles applied while developing HPX

As it 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 already exist, many for more than 30 years. All it takes is to gather them into a coherent approach. We'll highlight some of the derived principles we think to be crucial for defeating **SLOW**. Some of those are focused on high-performance computing, others are more general.

Focus on latency hiding instead of latency avoidance

It is impossible to design a system exposing zero latencies. In an effort to come as close as possible to this goal many optimizations are mainly targeted towards minimizing latencies. Examples for this can be seen everywhere, such as low latency network technologies like [InfiniBand¹⁵⁷](#), caching memory hierarchies in all modern processors, the constant optimization of existing [MPI¹⁵⁸](#) implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use [GPGPUs¹⁵⁹](#) 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.

¹⁵⁶ http://en.wikipedia.org/wiki/Mean_time_between_failures

¹⁵⁷ <http://en.wikipedia.org/wiki/InfiniBand>

¹⁵⁸ https://en.wikipedia.org/wiki/Message_Passing_Interface

¹⁵⁹ <http://en.wikipedia.org/wiki/GPGPU>

Embrace fine-grained parallelism instead of heavyweight threads

If we plan to hide latencies even for very short operations, such as fetching the contents of a memory cell from main memory (if it is not already cached), we need to have very lightweight threads with extremely short context switching times, optimally executable within one cycle. Granted, for mainstream architectures, this is not possible today (even if we already have special machines supporting this mode of operation, such as the [Cray XMT](#)¹⁶⁰). 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 \(TBB\)](#)¹⁶¹, [Microsoft Parallel Patterns Library \(PPL\)](#)¹⁶², [Cilk++](#)¹⁶³, and many others. The possibility to suspend a current task if some preconditions for its execution are not met (such as waiting for I/O or the result of a different task), seamlessly switching to any other task which can continue, and to reschedule the initial task after the required result has been calculated, which makes the implementation of latency hiding almost trivial.

Rediscover constraint-based synchronization to replace global barriers

The code we write today is riddled with implicit (and explicit) global barriers. By “global barriers,” we mean the synchronization of the control flow between several (very often all) threads (when using [OpenMP](#)¹⁶⁴) or processes ([MPI](#)¹⁶⁵). For instance, an implicit global barrier is inserted after each loop parallelized using [OpenMP](#)¹⁶⁶ as the system synchronizes the threads used to execute the different iterations in parallel. In [MPI](#)¹⁶⁷ each of the communication steps imposes an explicit barrier onto the execution flow as (often all) nodes have to be synchronized. Each of those barriers is like the eye of a needle the overall execution is forced to be squeezed through. Even minimal fluctuations in the execution times of the parallel threads (jobs) causes them to wait. Additionally, it is often only one of the executing threads that performs the actual reduce operation, which further impedes parallelism. A closer analysis of a couple of key algorithms used in science applications reveals that these global barriers are not always necessary. In many cases it is sufficient to synchronize a small subset of the threads. Any operation should proceed whenever the preconditions for its execution are met, and only those. Usually there is no need to wait for iterations of a loop to finish before you can continue calculating other things; all you need is to complete the iterations that produce the required results for the next operation. Good bye global barriers, hello constraint based synchronization! People have been trying to build this type of computing (and even computers) since the 1970s. The theory behind what they did is based on ideas around static and dynamic dataflow. There are certain attempts today to get back to those ideas and to incorporate them with modern architectures. For instance, a lot of work is being done in the area of constructing dataflow-oriented execution trees. Our results show that employing dataflow techniques in combination with the other ideas, as outlined herein, considerably improves scalability for many problems.

¹⁶⁰ http://en.wikipedia.org/wiki/Cray_XMT

¹⁶¹ <https://www.threadingbuildingblocks.org/>

¹⁶² <https://msdn.microsoft.com/en-us/library/dd492418.aspx>

¹⁶³ <https://software.intel.com/en-us/articles/intel-cilk-plus/>

¹⁶⁴ <https://openmp.org/wp/>

¹⁶⁵ https://en.wikipedia.org/wiki/Message_Passing_Interface

¹⁶⁶ <https://openmp.org/wp/>

¹⁶⁷ https://en.wikipedia.org/wiki/Message_Passing_Interface

Adaptive locality control instead of static data distribution

While this principle seems to be a given for single desktop or laptop computers (the operating system is your friend), it is everything but ubiquitous on modern supercomputers, which are usually built from a large number of separate nodes (i.e., Beowulf clusters), tightly interconnected by a high-bandwidth, low-latency network. Today's prevalent programming model for those is MPI, which does not directly help with proper data distribution, leaving it to the programmer to decompose the data to all of the nodes the application is running on. There are a couple of specialized languages and programming environments based on PGAS¹⁶⁸ (Partitioned Global Address Space) designed to overcome this limitation, such as Chapel¹⁶⁹, X10¹⁷⁰, UPC¹⁷¹, or Fortress¹⁷². However, all systems based on PGAS rely on static data distribution. This works fine as long as this static data distribution does not result in heterogeneous workload distributions or other resource utilization imbalances. In a distributed system these imbalances can be mitigated by migrating part of the application data to different localities (nodes). The only framework supporting (limited) migration today is Charm++¹⁷³. The first attempts towards solving related problem go back decades as well, a good example is the Linda coordination language¹⁷⁴. Nevertheless, none of the other mentioned systems support data migration today, which forces the users to either rely on static data distribution and live with the related performance hits or to implement everything themselves, which is very tedious and difficult. We believe that the only viable way to flexibly support dynamic and adaptive *locality* control is to provide a global, uniform address space to the applications, even on distributed systems.

Prefer moving work to the data over moving data to the work

For the best performance it seems obvious to minimize the amount of bytes transferred from one part of the system to another. This is true on all levels. At the lowest level we try to take advantage of processor memory caches, thus, minimizing memory latencies. Similarly, we try to amortize the data transfer time to and from GPGPUs¹⁷⁵ as much as possible. At high levels we try to minimize data transfer between different nodes of a cluster or between different virtual machines on the cloud. Our experience (well, it's almost common wisdom) shows that the amount of bytes necessary to encode a certain operation is very often much smaller than the amount of bytes encoding the data the operation is performed upon. Nevertheless, we still often transfer the data to a particular place where we execute the operation just to bring the data back to where it came from afterwards. As an example let's look at the way we usually write our applications for clusters using MPI. This programming model is all about data transfer between nodes. MPI is the prevalent programming model for clusters, and it is fairly straightforward to understand and to use. Therefore, we often write applications in a way that accommodates this model, centered around data transfer. These applications usually work well for smaller problem sizes and for regular data structures. The larger the amount of data we have to churn and the more irregular the problem domain becomes, the worse the overall machine utilization and the (strong) scaling characteristics become. While it is not impossible to implement more dynamic, data driven, and asynchronous applications using MPI, it is somewhat difficult to do so. At the same time, if we look at applications that prefer to execute the code close to the *locality* where the data was placed, i.e., utilizing active messages (for instance based on Charm++¹⁷⁶), we see better asynchrony, simpler application codes, and improved scaling.

¹⁶⁸ <https://www.pgas.org/>

¹⁶⁹ <https://chapel.cray.com/>

¹⁷⁰ <https://x10-lang.org/>

¹⁷¹ <https://upc.lbl.gov/>

¹⁷² <https://labs.oracle.com/projects/plrg/Publications/index.html>

¹⁷³ <https://charm.cs.uiuc.edu/>

¹⁷⁴ [http://en.wikipedia.org/wiki/Linda_\(coordination_language\)](http://en.wikipedia.org/wiki/Linda_(coordination_language))

¹⁷⁵ <http://en.wikipedia.org/wiki/GPGPU>

¹⁷⁶ <https://charm.cs.uiuc.edu/>

Favor message driven computation over message passing

Today's prevalently used programming model on parallel (multi-node) systems is MPI. It is based on message passing, as the name implies, which means that the receiver has to be aware of a message about to come in. Both codes, the sender and the receiver, have to synchronize in order to perform the communication step. Even the newer, asynchronous interfaces require explicitly coding the algorithms around the required communication scheme. As a result, everything but the most trivial MPI applications spends a considerable amount of time waiting for incoming messages, thus, causing starvation and latencies to impede full resource utilization. The more complex and more dynamic the data structures and algorithms become, the larger the adverse effects. The community discovered message-driven and data-driven methods of implementing algorithms a long time ago, and systems such as [Charm++¹⁷⁷](#) have already integrated active messages demonstrating the validity of the concept. Message-driven computation allows for sending messages without requiring the receiver to actively wait for them. Any incoming message is handled asynchronously and triggers the encoded action by passing along arguments and—possibly—continuations. *HPX* combines this scheme with work-queue based scheduling as described above, which allows the system to almost completely overlap any communication with useful work, thereby minimizing latencies.

2.6 Additional material

- 2-day workshop held at CSCS in 2016
 - Recorded lectures¹⁷⁸
 - Slides¹⁷⁹
- Tutorials repository¹⁸⁰
- STELLAR Group blog posts¹⁸¹
- Basic *HPX* recipes
 - Exporting a free function from a shared library which lives in a namespace, to use as Action¹⁸²
 - Turning a struct or class into a component and use it's methods¹⁸³
 - Creating and referencing components in hpx¹⁸⁴

2.7 Overview

HPX is organized into different sub-libraries and those in turn into modules. The libraries and modules are independent, with clear dependencies and no cycles. As an end-user, the use of these libraries is completely transparent. If you use e.g. `add_hpx_executable` to create a target in your project you will automatically get all modules as dependencies. See below for a list of the available libraries and modules. Currently these are nothing more than an internal grouping and do not affect usage. They cannot be consumed individually at the moment.

¹⁷⁷ <https://charm.cs.uiuc.edu/>

¹⁷⁸ <https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOnIJ0lFNjKze18>

¹⁷⁹ <https://github.com/STELLAR-GROUP/tutorials/tree/master/cscs2016>

¹⁸⁰ <https://github.com/STELLAR-GROUP/tutorials>

¹⁸¹ <http://stellar-group.org/blog/>

¹⁸² <https://gitlab.com/-/snippets/1821389>

¹⁸³ <https://gitlab.com/-/snippets/1822983>

¹⁸⁴ <https://gitlab.com/-/snippets/1828131>

2.7.1 Core modules

affinity

The affinity module contains helper functionality for mapping worker threads to hardware resources.

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

algorithms

The algorithms module exposes the full set of algorithms defined by the C++ standard. There is also partial support for C++ ranges.

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

allocator_support

This module provides utilities for allocators. It contains `hpx::util::internal_allocator` which directly forwards allocation calls to `jemalloc`. This utility is mainly useful on Windows.

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

asio

The asio module is a thin wrapper around the Boost.ASIO library, providing a few additional helper functions.

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

assertion

The assertion library implements the macros `HPX_ASSERT` and `HPX_ASSERT_MSG`. Those two macros can be used to implement assertions which are turned off during a release build.

By default, the location and function where the assert has been called from are displayed when the assertion fires. This behavior can be modified by using `hpx::assertion::set_assertion_handler`. When HPX initializes, it uses this function to specify a more elaborate assertion handler. If your application needs to customize this, it needs to do so before calling `hpx::hpx_init`, `hpx::hpx_main` or using the C-main wrappers.

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

async_base

The `async_base` module defines the basic functionality for spawning tasks on thread pools. This module does not implement any functionality on its own, but is extended by `async_local` and `modules_async_distributed` with implementations for the local and distributed cases.

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

async_combinators

This module contains combinators for futures. The `when_*` functions allow you to turn multiple futures into a single future which is ready when all, any, some, or each of the given futures are ready. The `wait_*` combinators are equivalent to the `when_*` functions except that they do not return a future.

The `split_future` combinator takes a single future of a container (e.g. `tuple`) and turns it into a container of futures.

See `lcos_local`, `synchronization`, and `async` for other synchronization facilities.

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

async_cuda

This library adds a simple API that enables the user to retrieve a future from a cuda stream. Typically, a user may launch one or more kernels and then get a future from the stream that will become ready when those kernels have completed. It is important to note that multiple kernels may be launched without fetching a future, and multiple futures may be obtained from the helper. Please refer to the unit tests and examples for further examples.

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

async_local

This module extends `async_base` to provide local implementations of `hpx::async`, `hpx::apply`, `hpx::sync`, and `hpx::dataflow`.

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

async_mpi

The MPI library is intended to simplify the process of integrating MPI based codes with the *HPX* runtime. Any MPI function that is asynchronous and uses an MPI_Request may be converted into an `hpx::future`. The syntax is designed to allow a simple replacement of the MPI call with a futurized async version that accepts an executor instead of a communicator, and returns a future instead of assigning a request. Typically, an MPI call of the form

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

becomes

```
hpx::future<int> f = hpx::async(executor, MPI_Isend, buf, count, datatype, rank, tag);
```

When the MPI operation is complete, the future will become ready. This allows communication to integrated cleanly with the rest of HPX, in particular the continuation style of programming may be used to build up more complex code. Consider the following example, that chains user processing, sends and receives using continuations...

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

// post an asynchronous receive using MPI_Irecv
hpx::future<int> f_recv = hpx::async(
    exec, MPI_Irecv, &data, rank, MPI_INT, rank_from, i);

// attach a continuation to run when the recv completes,
f_recv.then([=, &tokens, &counter] (auto&&)
```

(continues on next page)

(continued from previous page)

```
{
    // call an application specific function
    msg_recv(rank, size, rank_to, rank_from, tokens[i], i);

    // send a new message
    hpx::future<int> f_send = hpx::async(
        exec, MPI_Isend, &tokens[i], 1, MPI_INT, rank_to, i);

    // when that send completes
    f_send.then([=, &tokens, &counter] (auto&&)
    {
        // call an application specific function
        msg_send(rank, size, rank_to, rank_from, tokens[i], i);
    });
}
```

The example above makes use of `MPI_Isend` and `MPI_Irecv`, but *any* MPI function that uses requests may be futurized in this manner. The following is a (non exhaustive) list of MPI functions that *should* be supported, though not all have been tested at the time of writing (please report any problems to the issue tracker).

```
int MPI_Isend(...);
int MPI_Ibsend(...);
int MPI_Issend(...);
int MPI_Irsend(...);
int MPI_Irecv(...);
int MPI_Imrecv(...);
int MPI_Ibarrier(...);
int MPI_Ibcast(...);
int MPI_Igather(...);
int MPI_Igatherv(...);
int MPI_Iscatter(...);
int MPI_Iscatterv(...);
int MPI_Iallgather(...);
int MPI_Iallgatherv(...);
int MPI_Ialltoall(...);
int MPI_Ialltoallv(...);
int MPI_Ialltoallw(...);
int MPI_Ireduce(...);
int MPI_Iallreduce(...);
int MPI_Ireduce_scatter(...);
int MPI_Ireduce_scatter_block(...);
int MPI_Iscan(...);
int MPI_Iexscan(...);
int MPI_Ineighbor_allgather(...);
int MPI_Ineighbor_allgatherv(...);
int MPI_Ineighbor_alltoall(...);
int MPI_Ineighbor_alltoallv(...);
int MPI_Ineighbor_alltoallw(...);
```

Note that the `HPX mpi` futurization wrapper should work with *any* asynchronous `MPI` call, as long as the function signature has the last two arguments `MPI_xxx(..., MPI_Comm comm, MPI_Request *request)` - internally these two parameters will be substituted by the executor and future data parameters that are supplied by template instantiations inside the `hpx::mpi` code.

See the API reference of this module for more details.

batch_environments

This module allows for the detection of execution as batch jobs, a series of programs executed without user intervention. All data is preselected and will be executed according to preset parameters, such as date or completion of another task. Batch environments are especially useful for executing repetitive tasks.

HPX supports the creation of batch jobs through the Portable Batch System (PBS) and SLURM.

For more information on batch environments, see *Running on batch systems* and the *API reference* for the module.

cache

This module provides two cache data structures:

- `hpx::util::cache::local_cache`
- `hpx::util::cache::lru_cache`

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

command_line_handling_local

TODO: High-level description of the module.

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

concepts

This module provides helpers for emulating concepts. It provides the following macros:

- `HPX_CONCEPT_REQUIRES`
- `HPX_HAS_MEMBER_XXX_TRAIT_DEF`
- `HPX_HAS_XXX_TRAIT_DEF`

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

concurrency

This module provides concurrency primitives useful for multi-threaded programming such as:

- `hpx::util::barrier`
- `hpx::util::cache_line_data` and `hpx::util::cache_aligned_data`: wrappers for aligning and padding data to cache lines.
- various lockfree queue data structures

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

config

The config module contains various configuration options, typically hidden behind macros that choose the correct implementation based on the compiler and other available options.

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

config_registry

The config_registry module is a low level module providing helper functionality for registering configuration entries to a global registry from other modules. The `hpx::config_registry::add_module_config` function is used to add configuration options, and `hpx::config_registry::get_module_configs` can be used to retrieve configuration entries registered so far. `add_module_config_helper` can be used to register configuration entries through static global options.

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

coroutines

The coroutines module provides coroutine (user-space thread) implementations for different platforms.

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

datastructures

The datastructures module provides basic data structures (typically provided for compatibility with older C++ standards):

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

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

debugging

This module provides helpers for demangling symbol names.

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

errors

This module provides support for exceptions and error codes:

- `hpx::exception`
- `hpx::error_code`
- `hpx::error`

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

execution

This library implements executors and execution policies for use with parallel algorithms and other facilities related to managing the execution of tasks.

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

execution_base

The basic execution module is the main entry point to implement parallel and concurrent operations. It is modeled after P0443¹⁸⁵ with some additions and implementations for the described concepts. Most notably, it provides an abstraction for execution resources, execution contexts and execution agents in such a way, that it provides customization points that those aforementioned concepts can be replaced and combined with ease.

For that purpose, three virtual base classes are provided to be able to provide implementations with different properties:

- **resource_base**: This is the abstraction for execution resources, that is for example CPU cores or an accelerator.
- **context_base**: An execution context uses execution resources and is able to spawn new execution agents, as new threads of executions on the available resources.
- **agent_base**: The execution agent represents the thread of execution, and can be used to yield, suspend, resume or abort a thread of execution.

executors

The executors module exposes executors and execution policies. Most importantly, it exposes the following classes and constants:

- `hpx::execution::sequenced_executor`
- `hpx::execution::parallel_executor`
- `hpx::execution::sequenced_policy`
- `hpx::execution::parallel_policy`
- `hpx::execution::parallel_unsequenced_policy`
- `hpx::execution::sequenced_task_policy`
- `hpx::execution::parallel_task_policy`
- `hpx::execution::seq`
- `hpx::execution::par`
- `hpx::execution::par_unseq`
- `hpx::execution::task`

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

¹⁸⁵ <http://wg21.link/p0443>

filesystem

This module provides a compatibility layer for the C++17 filesystem library. If the filesystem library is available this module will simply forward its contents into the `hpx::filesystem` namespace. If the library is not available it will fall back to Boost.Filesystem instead.

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

format

The format module exposes the `format` and `format_to` functions for formatting strings.

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

functional

This module provides function wrappers and helpers for managing functions and their arguments.

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

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

futures

This module defines the `hpx::future` and `hpx::shared_future` classes corresponding to the C++ standard library classes `std::future` and `std::shared_future`. Note that the specializations of `hpx::future::then` for executors and execution policies are defined in the *execution* module.

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

hardware

The hardware module abstracts away hardware specific details of timestamps and CPU features.

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

hashing

The hashing module provides two hashing implementations:

- `hpx::util::fibhash`
- `hpx::util::jenkins_hash`

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

include_local

This module provides no functionality in itself. Instead it provides headers that group together other headers that often appear together. This module provides local-only headers.

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

ini

TODO: High-level description of the module.

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

init_runtime_local

TODO: High-level description of the module.

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

io_service

This module provides an abstraction over Boost.ASIO, combining multiple `asio::io_contexts` into a single pool. `hpx::util::io_service_pool` provides a simple pool of `asio::io_contexts` with an API similar to `asio::io_context`. `hpx::threads::detail::io_service_thread_pool` wraps `hpx::util::io_service_pool` into an interface derived from `hpx::threads::detail::thread_pool_base`.

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

iterator_support

This module provides helpers for iterators. It provides `hpx::util::iterator_facade` and `hpx::util::iterator_adaptor` for creating new iterators, and the trait `hpx::util::is_iterator` along with more specific iterator traits.

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

itt_notify

This module provides support for profiling with Intel VTune¹⁸⁶.

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

lcos_local

This module provides the following local *LCOs*:

- `hpx::lcos::local::and_gate`
- `hpx::lcos::local::channel`
- `hpx::lcos::local::one_element_channel`
- `hpx::lcos::local::receive_channel`
- `hpx::lcos::local::send_channel`
- `hpx::lcos::local::guard`
- `hpx::lcos::local::guard_set`
- `hpx::lcos::local::run_guarded`
- `hpx::lcos::local::conditional_trigger`
- `hpx::lcos::local::packaged_task`
- `hpx::lcos::local::promise`
- `hpx::lcos::local::receive_buffer`
- `hpx::lcos::local::trigger`

See *lcos_distributed* for distributed LCOs. Basic synchronization primitives for use in HPX threads can be found in *synchronization*. *async_combinators* contains useful utility functions for combining futures.

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

lock_registration

This module contains functionality for registering locks to detect when they are locked and unlocked on different threads.

See the API reference of this module for more details.

¹⁸⁶ <https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html>

logging

This module provides useful macros for logging information.

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

memory

Part of this module is a forked version of boost::intrusive_ptr from Boost.SmartPtr.

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

mpi_base

This module provides helper functionality for detecting MPI environments.

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

pack_traversal

This module exposes the basic functionality for traversing various packs, both synchronously and asynchronously: `hpx::util::traverse_pack` and `hpx::util::traverse_pack_async`. It also exposes the higher level functionality of unwrapping nested futures: `hpx::util::unwrap` and its function object form `hpx::util::functional::unwrap`.

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

plugin

This module provides base utilities for creating plugins.

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

prefix

This module provides utilities for handling the prefix of an *HPX* application, i.e. the paths used for searching components and plugins.

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

preprocessor

This library contains useful preprocessor macros:

- `HPX_PP_CAT`
- `HPX_PP_EXPAND`
- `HPX_PP_NARGS`
- `HPX_PP_STRINGIZE`
- `HPX_PP_STRIP_PARENS`

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

program_options

The module `program_options` is a direct fork of the `Boost.ProgramOptions` library (Boost V1.70.0). For more information about this library please see [here¹⁸⁷](#). In order to be included as an *HPX* module, the `Boost.ProgramOptions` library has been moved to the namespace `hpx::program_options`. We have also replaced all Boost facilities the library depends on with either the equivalent facilities from the standard library or from *HPX*. As a result, the *HPX* `program_options` module is fully interface compatible with `Boost.ProgramOptions` (sans the `hpx` namespace and the `#include <hpx/modules/program_options.hpp>` changes that need to be applied to all code relying on this library).

All credit goes to Vladimir Prus, the author of the excellent `Boost.ProgramOptions` library. All bugs have been introduced by us.

See the *API reference* of the module for more details.

properties

This module implements the `prefer` customization point for properties in terms of [P2220¹⁸⁸](#). This differs from [P1393¹⁸⁹](#) in that it relies fully on `tag_invoke` overloads and fewer base customization points. Actual properties are defined in modules. All functionality is experimental and can be accessed through the `hpx::experimental` namespace.

See the *API reference* of this module for more details.

resiliency

In *HPX*, a program failure is a manifestation of a failing task. This module exposes several APIs that allow users to manage failing tasks in a convenient way by either replaying a failed task or by replicating a specific task.

Task replay is analogous to the Checkpoint/Rollback mechanism found in conventional execution models. The key difference being localized fault detection. When the runtime detects an error, it replays the failing task as opposed to completely rolling back the entire program to the previous checkpoint.

Task replication is designed to provide reliability enhancements by replicating a set of tasks and evaluating their results to determine a consensus among them. This technique is most effective in situations where there are few tasks in the critical path of the DAG which leaves the system underutilized or where hardware or software failures may result in an incorrect result instead of an error. However, the drawback of this method is the additional computational cost incurred by repeating a task multiple times.

The following API functions are exposed:

- `hpx::resiliency::experimental::async_replay`: This version of task replay will catch user-defined exceptions and automatically reschedule the task N times before throwing an `hpx::resiliency::experimental::abort_replay_exception` if no task is able to complete execution without an exception.
- `hpx::resiliency::experimental::async_replay_validate`: This version of replay adds an argument to `async replay` which receives a user-provided validation function to test the result of the task against. If the task's output is validated, the result is returned. If the output fails the check or an exception is thrown, the task is replayed until no errors are encountered or the number of specified retries has been exceeded.
- `hpx::resiliency::experimental::async_replicate`: This is the most basic implementation of the task replication. The API returns the first result that runs without detecting any errors.

¹⁸⁷ https://www.boost.org/doc/libs/1_70_0/doc/html/program_options.html

¹⁸⁸ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2220r0.pdf>

¹⁸⁹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1393r0.html>

- *hpx::resiliency::experimental::async_replicate_validate*: This API additionally takes a validation function which evaluates the return values produced by the threads. The first task to compute a valid result is returned.
- *hpx::resiliency::experimental::async_replicate_vote*: This API adds a vote function to the basic replicate function. Many hardware or software failures are silent errors which do not interrupt program flow. In order to detect errors of this kind, it is necessary to run the task several times and compare the values returned by every version of the task. In order to determine which return value is “correct”, the API allows the user to provide a custom consensus function to properly form a consensus. This voting function then returns the “correct” answer.
- *hpx::resiliency::experimental::async_replicate_vote_validate*: This combines the features of the previously discussed replicate set. Replicate vote validate allows a user to provide a validation function to filter results. Additionally, as described in replicate vote, the user can provide a “voting function” which returns the consensus formed by the voting logic.
- *hpx::resiliency::experimental::dataflow_replay*: This version of dataflow replay will catch user-defined exceptions and automatically reschedules the task N times before throwing an *hpx::resiliency::experimental::abort_replay_exception* if no task is able to complete execution without an exception. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replay_validate*: This version of replay adds an argument to dataflow replay which receives a user-provided validation function to test the result of the task against. If the task’s output is validated, the result is returned. If the output fails the check or an exception is thrown, the task is replayed until no errors are encountered or the number of specified retries have been exceeded. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate*: This is the most basic implementation of the task replication. The API returns the first result that runs without detecting any errors. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate_validate*: This API additionally takes a validation function which evaluates the return values produced by the threads. The first task to compute a valid result is returned. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate_vote*: This API adds a vote function to the basic replicate function. Many hardware or software failures are silent errors which do not interrupt program flow. In order to detect errors of this kind, it is necessary to run the task several times and compare the values returned by every version of the task. In order to determine which return value is “correct”, the API allows the user to provide a custom consensus function to properly form a consensus. This voting function then returns the “correct” answer. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate_vote_validate*: This combines the features of the previously discussed replicate set. Replicate vote validate allows a user to provide a validation function to filter results. Additionally, as described in replicate vote, the user can provide a “voting function” which returns the consensus formed by the voting logic. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.

See the *API reference* of the module for more details.

resource_partitioner

The `resource_partitioner` module defines `hpx::resource::partitioner`, the class used by the runtime and users to partition available hardware resources into thread pools. See *Using the resource partitioner* for more details on using the resource partitioner in applications.

See the *API reference* of this module for more details.

runtime_configuration

This module handles the configuration options required by the runtime.

See the *API reference* of this module for more details.

runtime_local

TODO: High-level description of the library.

See the *API reference* of this module for more details.

schedulers

This module provides schedulers used by thread pools in the `thread_pools` module. There are currently three main schedulers:

- `hpx::threads::policies::local_priority_queue_scheduler`
- `hpx::threads::policies::static_priority_queue_scheduler`
- `hpx::threads::policies::shared_priority_queue_scheduler`

Other schedulers are specializations or variations of the above schedulers. See the examples of the `resource_partitioner` module for examples of specifying a custom scheduler for a thread pool.

See the *API reference* of this module for more details.

serialization

This module provides serialization primitives and support for all built-in types as well as all C++ Standard Library collection and utility types. This list is extended by *HPX* vocabulary types with proper support for global reference counting. *HPX*'s mode of serialization is derived from [Boost's serialization model¹⁹⁰](#) and, as such, is mostly interface compatible with its Boost counterpart.

The purest form of serializing data is to copy the content of the payload bit by bit; however, this method is impractical for generic C++ types, which might be composed of more than just regular built-in types. Instead, *HPX*'s approach to serialization is derived from the Boost Serialization library, and is geared towards allowing the programmer of a given class explicit control and syntax of what to serialize. It is based on operator overloading of two special archive types that hold a buffer or stream to store the serialized data and is responsible for dispatching the serialization mechanism to the intrusive or non-intrusive version. The serialization process is recursive. Each member that needs to be serialized must be specified explicitly. The advantage of this approach is that the serialization code is written in C++ and leverages all necessary programming techniques. The generic, user-facing interface allows for effective application of the serialization process without obstructing the algorithms that need special code for packing and unpacking. It also allows for optimizations in the implementation of the archives.

See the *API reference* of the module for more details.

¹⁹⁰ https://www.boost.org/doc/libs/1_72_0/libs/serialization/doc/index.html

static_reinit

This module provides a simple wrapper around static variables that can be reinitialized.

See the *API reference* of this module for more details.

string_util

This module contains string utilities inspired by the Boost string algorithms library.

See the *API reference* of this module for more details.

synchronization

This module provides synchronization primitives which should be used rather than the C++ standard ones in *HPX* threads:

- `hpx::lcos::local::barrier`
- `hpx::lcos::local::condition_variable`
- `hpx::lcos::local::counting_semaphore`
- `hpx::lcos::local::event`
- `hpx::lcos::local::latch`
- `hpx::lcos::local::mutex`
- `hpx::lcos::local::no_mutex`
- `hpx::lcos::local::once_flag`
- `hpx::lcos::local::recursive_mutex`
- `hpx::lcos::local::shared_mutex`
- `hpx::lcos::local::sliding_semaphore`
- `hpx::lcos::local::spinlock` (`std::mutex` compatible spinlock)
- `hpx::lcos::local::spinlock_no_backoff` (`boost::mutex` compatible spinlock)
- `hpx::lcos::local::spinlock_pool`

See `lcos_local`, `async_combinators`, and `async` for higher level synchronization facilities.

See the *API reference* of this module for more details.

tag_invoke

TODO: High-level description of the module.

See the *API reference* of this module for more details.

testing

The testing module contains useful macros for testing. The results of tests can be printed with `hpx::util::report_errors`. The following macros are provided:

- `HPX_TEST`
- `HPX_TEST_MSG`
- `HPX_TEST_EQ`
- `HPX_TEST_NEQ`
- `HPX_TEST_LT`
- `HPX_TEST_LTE`
- `HPX_TEST_RANGE`
- `HPX_TEST_EQ_MSG`
- `HPX_TEST_NEQ_MSG`
- `HPX_SANITY`
- `HPX_SANITY_MSG`
- `HPX_SANITY_EQ`
- `HPX_SANITY_NEQ`
- `HPX_SANITY_LT`
- `HPX_SANITY_LTE`
- `HPX_SANITY_RANGE`
- `HPX_SANITY_EQ_MSG`

See the *API reference* of the module for more details.

thread_pool_util

This module contains helper functions for asynchronously suspending and resuming thread pools and their worker threads.

See the *API reference* of this module for more details.

thread_pools

This module defines the thread pools and utilities used by the *HPX* runtime. The only thread pool implementation provided by this module is `hpx::threads::detail::scheduled_thread_pool`, which is derived from `hpx::threads::detail::thread_pool_base` defined in the *threading_base* module.

See the *API reference* of this module for more details.

thread_support

This module provides miscellaneous utilities for threading and concurrency.

See the *API reference* of the module for more details.

threading

This module provides the equivalents of `std::thread` and `std::jthread` for lightweight *HPX* threads:

- `hpx::thread`
- `hpx::jthread`

See the *API reference* of this module for more details.

threading_base

This module contains the base class definition required for threads. The base class `hpx::threads::thread_data` is inherited by two specializations for stackful and stackless threads: `hpx::threads::thread_data_stackful` and `hpx::threads::thread_data_stackless`. In addition, the module defines the base classes for schedulers and thread pools: `hpx::threads::policies::scheduler_base` and `hpx::threads::thread_pool_base`.

See the *API reference* of this module for more details.

thread_manager

This module defines the `hpx::threads::threadmanager` class. This is used by the runtime to manage the creation and destruction of thread pools. The `resource_partitioner` module handles the partitioning of resources into thread pools, but not the creation of thread pools.

See the *API reference* of this module for more details.

timed_execution

This module provides extensions to the executor interfaces defined in the `execution` module that allow timed submission of tasks on thread pools (at or after a specified time).

See the *API reference* of this module for more details.

timing

This module provides the timing utilities (clocks and timers).

See the *API reference* of the module for more details.

topology

This module provides the class `hpx::threads::topology` which represents the hardware resources available on a node. The class is a light wrapper around the [Portable Hardware Locality \(HWLOC\)](#)¹⁹¹ library. The `hpx::threads::cpu_mask` is a small companion class that represents a set of resources on a node.

See the *API reference* of the module for more details.

type_support

This module provides helper facilities related to types.

See the *API reference* of the module for more details.

util

The util module provides miscellaneous standalone utilities.

See the *API reference* of the module for more details.

version

This module macros and functions for accessing version information about *HPX* and its dependencies.

See the *API reference* of this module for more details.

2.7.2 Main HPX modules

actions

TODO: High-level description of the library.

See the *API reference* of this module for more details.

actions_base

TODO: High-level description of the library.

See the *API reference* of this module for more details.

agas

TODO: High-level description of the module.

See the *API reference* of this module for more details.

¹⁹¹ <https://www.open-mpi.org/projects/hwloc/>

agas_base

This module holds the implementation of the four AGAS services: primary namespace, locality namespace, component namespace, and symbol namespace.

See the *API reference* of this module for more details.

async_colocated

TODO: High-level description of the module.

See the *API reference* of this module for more details.

async

This module contains functionality for asynchronously launching work on remote localities: `hpx::async`, `hpx::apply`. This module extends the local-only functions in `libs_async_local`.

See the API reference of this module for more details.

checkpoint

A common need of users is to periodically backup an application. This practice provides resiliency and potential restart points in code. *HPX* utilizes the concept of a `checkpoint` to support this use case.

Found in `hpx/util/checkpoint.hpp`, `checkpoints` are defined as objects that hold a serialized version of an object or set of objects at a particular moment in time. This representation can be stored in memory for later use or it can be written to disk for storage and/or recovery at a later point. In order to create and fill this object with data, users must use a function called `save_checkpoint`. In code the function looks like this:

```
hpx::future<hpx::util::checkpoint> hpx::util::save_checkpoint(a, b, c, ...);
```

`save_checkpoint` takes arbitrary data containers, such as `int`, `double`, `float`, `vector`, and `future`, and serializes them into a newly created `checkpoint` object. This function returns a `future` to a `checkpoint` containing the data. Here's an example of a simple use case:

```
using hpx::util::checkpoint;
using hpx::util::save_checkpoint;

std::vector<int> vec{1,2,3,4,5};
hpx::future<checkpoint> save_checkpoint(vec);
```

Once the future is ready, the `checkpoint` object will contain the `vector` `vec` and its five elements.

`prepare_checkpoint` takes arbitrary data containers (same as for `save_checkpoint`), , such as `int`, `double`, `float`, `vector`, and `future`, and calculates the necessary buffer space for the `checkpoint` that would be created if `save_checkpoint` was called with the same arguments. This function returns a `future` to a `checkpoint` that is appropriately initialized. Here's an example of a simple use case:

```
using hpx::util::checkpoint;
using hpx::util::prepare_checkpoint;

std::vector<int> vec{1,2,3,4,5};
hpx::future<checkpoint> prepare_checkpoint(vec);
```

Once the future is ready, the checkpoint object will be initialized with an appropriately sized internal buffer.

It is also possible to modify the launch policy used by `save_checkpoint`. This is accomplished by passing a launch policy as the first argument. It is important to note that passing `hpx::launch::sync` will cause `save_checkpoint` to return a `checkpoint` instead of a `future` to a `checkpoint`. All other policies passed to `save_checkpoint` will return a `future` to a `checkpoint`.

Sometimes `checkpoint`s must be declared before they are used. `save_checkpoint` allows users to move pre-created `checkpoint`s into the function as long as they are the first container passing into the function (In the case where a launch policy is used, the `checkpoint` will immediately follow the launch policy). An example of these features can be found below:

```
char character = 'd';
int integer = 10;
float flt = 10.01f;
bool boolean = true;
std::string str = "I am a string of characters";
std::vector<char> vec(str.begin(), str.end());
checkpoint archive;

// Test 1
// test basic functionality
hpx::shared_future<checkpoint> f_archive = save_checkpoint(
    std::move(archive), character, integer, flt, boolean, str, vec);
```

Once users can create `checkpoints` they must now be able to restore the objects they contain into memory. This is accomplished by the function `restore_checkpoint`. This function takes a `checkpoint` and fills its data into the containers it is provided. It is important to remember that the containers must be ordered in the same way they were placed into the `checkpoint`. For clarity see the example below:

```
char character2;
int integer2;
float flt2;
bool boolean2;
std::string str2;
std::vector<char> vec2;

restore_checkpoint(data, character2, integer2, flt2, boolean2, str2, vec2);
```

The core utility of `checkpoint` is in its ability to make certain data persistent. Often, this means that the data needs to be stored in an object, such as a file, for later use. *HPX* has two solutions for these issues: stream operator overloads and access iterators.

HPX contains two stream overloads, `operator<<` and `operator>>`, to stream data out of and into `checkpoint`. Here is an example of the overloads in use below:

```
double a9 = 1.0, b9 = 1.1, c9 = 1.2;
std::ofstream test_file_9("test_file_9.txt");
hpx::future<checkpoint> f_9 = save_checkpoint(a9, b9, c9);
test_file_9 << f_9.get();
test_file_9.close();

double a9_1, b9_1, c9_1;
std::ifstream test_file_9_1("test_file_9.txt");
checkpoint archive9;
test_file_9_1 >> archive9;
restore_checkpoint(archive9, a9_1, b9_1, c9_1);
```

This is the primary way to move data into and out of a `checkpoint`. It is important to note, however, that users

should be cautious when using a stream operator to load data and another function to remove it (or vice versa). Both `operator<<` and `operator>>` rely on a `.write()` and a `.read()` function respectively. In order to know how much data to read from the `std::istream`, the `operator<<` will write the size of the checkpoint before writing the checkpoint data. Correspondingly, the `operator>>` will read the size of the stored data before reading the data into a new instance of `checkpoint`. As long as the user employs the `operator<<` and `operator>>` to stream the data, this detail can be ignored.

Important: Be careful when mixing `operator<<` and `operator>>` with other facilities to read and write to a checkpoint. `operator<<` writes an extra variable, and `operator>>` reads this variable back separately. Used together the user will not encounter any issues and can safely ignore this detail.

Users may also move the data into and out of a checkpoint using the exposed `.begin()` and `.end()` iterators. An example of this use case is illustrated below.

```
std::ofstream test_file_7("checkpoint_test_file.txt");
std::vector<float> vec7{1.02f, 1.03f, 1.04f, 1.05f};
hpx::future<checkpoint> fut_7 = save_checkpoint(vec7);
checkpoint archive7 = fut_7.get();
std::copy(archive7.begin(),      // Write data to ofstream
          archive7.end(),     // ie. the file
          std::ostream_iterator<char>(test_file_7));
test_file_7.close();

std::vector<float> vec7_1;
std::vector<char> char_vec;
std::ifstream test_file_7_1("checkpoint_test_file.txt");
if (test_file_7_1)
{
    test_file_7_1.seekg(0, test_file_7_1.end);
    auto length = test_file_7_1.tellg();
    test_file_7_1.seekg(0, test_file_7_1.beg);
    char_vec.resize(length);
    test_file_7_1.read(char_vec.data(), length);
}
checkpoint archive7_1(std::move(char_vec));      // Write data to checkpoint
restore_checkpoint(archive7_1, vec7_1);
```

Checkpointing components

`save_checkpoint` and `restore_checkpoint` are also able to store components inside checkpoints. This can be done in one of two ways. First a client of the component can be passed to `save_checkpoint`. When the user wishes to resurrect the component she can pass a client instance to `restore_checkpoint`.

This technique is demonstrated below:

```
// Try to checkpoint and restore a component with a client
std::vector<int> vec3{10, 10, 10, 10, 10};

// Create a component instance through client constructor
data_client D(hpx::find_here(), std::move(vec3));
hpx::future<checkpoint> f3 = save_checkpoint(D);

// Create a new client
data_client E;
```

(continues on next page)

(continued from previous page)

```
// Restore server inside client instance
restore_checkpoint(f3.get(), E);
```

The second way a user can save a component is by passing a `shared_ptr` to the component to `save_checkpoint`. This component can be resurrected by creating a new instance of the component type and passing a `shared_ptr` to the new instance to `restore_checkpoint`.

This technique is demonstrated below:

```
// test checkpoint a component using a shared_ptr
std::vector<int> vec{1, 2, 3, 4, 5};
data_client A(hpx::find_here(), std::move(vec));

// Checkpoint Server
hpx::id_type old_id = A.get_id();

hpx::future<std::shared_ptr<data_server>> f_a_ptr =
    hpx::get_ptr<data_server>(A.get_id());
std::shared_ptr<data_server> a_ptr = f_a_ptr.get();
hpx::future<checkpoint> f = save_checkpoint(a_ptr);
auto&& data = f.get();

// test prepare_checkpoint API
checkpoint c = prepare_checkpoint(hpx::launch::sync, a_ptr);
HPX_TEST(c.size() == data.size());

// Restore Server
// Create a new server instance
std::shared_ptr<data_server> b_server;
restore_checkpoint(data, b_server);
```

checkpoint_base

The `checkpoint_base` module contains lower level facilities that wrap simple check-pointing capabilities. This module does not implement special handling for futures or components, but simply serializes all arguments to or from a given container.

This module exposes the `hpx::util::save_checkpoint_data`, `hpx::util::restore_checkpoint_data`, and `hpx::util::prepare_checkpoint_data` APIs. These functions encapsulate the basic serialization functionalities necessary to save/restore a variadic list of arguments to/from a given data container.

See the *API reference* of this module for more details.

collectives

The collectives module exposes a set of distributed collective operations. Those can be used to exchange data between participating sites in a coordinated way. At this point the module exposes the following collective primitives:

- `hpx::collectives::all_gather`: receives a set of values from all participating sites.
- `hpx::collectives::all_reduce`: performs a reduction on data from each participating site to each participating site.
- `hpx::collectives::all_to_all`: each participating site provides its element of the data to collect while all participating sites receive the data from every other site.

- `hpx::collectives::broadcast_to` and `hpx::collectives::broadcast_from`: performs a broadcast operation from a root site to all participating sites.
- `cpp:func:hpx::collectives::exclusive_scan` performs an exclusive scan operation on a set of values received from all call sites operating on the given base name.
- `hpx::collectives::gather_here` and `hpx::collectives::gather_there`: gathers values from all participating sites.
- `cpp:func:hpx::collectives::inclusive_scan` performs an inclusive scan operation on a set of values received from all call sites operating on the given base name.
- `hpx::collectives::reduce_here` and `hpx::collectives::reduce_there`: performs a reduction on data from each participating site to a root site.
- `hpx::collectives::scatter_to` and `hpx::collectives::scatter_from`: receives an element of a set of values operating on the given base name.
- `hpx::lcos::broadcast`: performs a given action on all given global identifiers.
- `hpx::lcos::barrier`: distributed barrier.
- `hpx::lcos::fold`: performs a fold with a given action on all given global identifiers.
- `hpx::lcos::latch`: distributed latch.
- `hpx::lcos::reduce`: performs a reduction on data from each given global identifiers.
- `hpx::lcos::spmd_block`: performs the same operation on a local image while providing handles to the other images.

See the *API reference* of the module for more details.

command_line_handling

The command_line_handling module defines and handles the command-line options required by the *HPX* runtime, combining them with configuration options defined by the *runtime_configuration* module. The actual parsing of command line options is handled by the *program_options* module.

See the *API reference* of the module for more details.

components

TODO: High-level description of the module.

See the *API reference* of this module for more details.

components_base

TODO: High-level description of the library.

See the *API reference* of this module for more details.

compute

The compute module provides utilities for handling task and memory affinity on host systems.

See the *API reference* of the module for more details.

distribution_policies

TODO: High-level description of the module.

See the *API reference* of this module for more details.

executors_distributed

This module provides the executor `hpx::parallel::execution::distribution_policy_executor`. It allows one to create work that is implicitly distributed over multiple localities.

See the *API reference* of this module for more details.

include

This module provides no functionality in itself. Instead it provides headers that group together other headers that often appear together.

See the *API reference* of this module for more details.

init_runtime

TODO: High-level description of the library.

See the *API reference* of this module for more details.

lcos_distributed

This module contains distributed *LCOs*. Currently the only LCO provided is `:cpp:class:hpx::lcos::channel`, a construct for sending values from one *locality* to another. See `libs_lcos_local` for local LCOs.

See the *API reference* of this module for more details.

naming

TODO: High-level description of the module.

See the *API reference* of this module for more details.

naming_base

This module provides a forward declaration of *address_type*, *component_type* and *invalid_locality_id*.

See the *API reference* of this module for more details.

parcelport_libfabric

TODO: High-level description of the module.

See the *API reference* of this module for more details.

parcelport_mpi

TODO: High-level description of the module.

See the *API reference* of this module for more details.

parcelport_tcp

TODO: High-level description of the module.

See the *API reference* of this module for more details.

parcelset

TODO: High-level description of the module.

See the *API reference* of this module for more details.

parcelset_base

TODO: High-level description of the module.

See the *API reference* of this module for more details.

performance_counters

This module provides the basic functionality required for defining performance counters. See *Performance counters* for more information about performance counters.

See the *API reference* of this module for more details.

plugin_factories

TODO: High-level description of the module.

See the *API reference* of this module for more details.

resiliency_distributed

Software resiliency features of HPX were introduced in the *resiliency module*. This module extends the APIs to run on distributed-memory systems allowing the user to invoke the failing task on other localities at runtime. This is useful in cases where a node is identified to fail more often (e.g., for certain ALU computes) as the task can now be replayed or replicated among different localities. The API exposed allows for an easy integration with the local only resiliency APIs as well.

Distributed software resilience APIs have a similar function signature and lives under the same namespace of `hpx::resiliency::experimental`. The difference arises in the formal parameters where distributed APIs takes the localities as the first argument, and an action as opposed to a function or a function object. The localities signify the order in which the API will either schedule (in case of Task Replay) tasks in a round robin fashion or replicate the tasks onto the list of localities.

The list of APIs exposed by distributed resiliency modules is the same as those defined in *local resiliency module*.

See the *API reference* of this module for more details.

runtime_components

TODO: High-level description of the module.

See the *API reference* of this module for more details.

runtime_distributed

TODO: High-level description of the module.

See the *API reference* of this module for more details.

segmented_algorithms

Segmented algorithms extend the usual parallel *algorithms* by providing overloads that work with distributed containers, such as partitioned vectors.

See the *API reference* of the module for more details.

statistics

This module provide some statistics utilities like rolling min/max and histogram.

See the *API reference* of the module for more details.

2.8 API reference

HPX follows a versioning scheme with three numbers: `major.minor.patch`. We guarantee no breaking changes in the API for patch releases. Minor releases may remove or break existing APIs, but only after a deprecation period of at least two minor releases. In rare cases do we outright remove old and unused functionality without a deprecation period.

We do not provide any ABI compatibility guarantees between any versions, debug and release builds, and builds with different C++ standards.

The public API of *HPX* is presented below. Clicking on a name brings you to the full documentation for the class or function. Including the header specified in a heading brings in the features listed under that heading.

Note: Names listed here are guaranteed stable with respect to semantic versioning. However, at the moment the list is incomplete and certain unlisted features are intended to be in the public API. While we work on completing the list, if you're unsure about whether a particular unlisted name is part of the public API you can get into contact with us or open an issue and we'll clarify the situation.

2.8.1 Public API

All names below are also available in the top-level `hpx` namespace unless otherwise noted. The names in `hpx` should be preferred. The names in sub-namespaces will eventually be removed.

Header `hpx/algorithm.hpp`

This header includes *Header hpx/local/algorithm.hpp* and contains overloads of the algorithms for segmented iterators.

Header `hpx/local/algorithm.hpp`

Corresponds to the C++ standard library header `algorithm`¹⁹². See *Using parallel algorithms* for more information about the parallel algorithms.

Classes

- `hpx::parallel::v2::reduction`
- `hpx::parallel::v2::induction`

Functions

- `hpx::adjacent_find`
- `hpx::all_of`
- `hpx::any_of`
- `hpx::copy`
- `hpx::copy_if`

¹⁹² <http://en.cppreference.com/w/cpp/header/algorithm>

- `hpx::copy_n`
- `hpx::count`
- `hpx::count_if`
- `hpx::ends_with`
- `hpx::equal`
- `hpx::fill`
- `hpx::fill_n`
- `hpx::find`
- `hpx::find_end`
- `hpx::find_first_of`
- `hpx::find_if`
- `hpx::find_if_not`
- `hpx::for_each`
- `hpx::for_each_n`
- `hpx::generate`
- `hpx::generate_n`
- `hpx::includes`
- `hpx::inplace_merge`
- `hpx::is_heap`
- `hpx::is_heap_until`
- `hpx::is_partitioned`
- `hpx::is_sorted`
- `hpx::is_sorted_until`
- `hpx::lexicographical_compare`
- `hpx::make_heap`
- `hpx::max_element`
- `hpx::merge`
- `hpx::min_element`
- `hpx::minmax_element`
- `hpx::parallel::v1::mismatch`
- `hpx::move`
- `hpx::none_of`
- `hpx::nth_element`
- `hpx::partial_sort`
- `hpx::partial_sort_copy`
- `hpx::partition`

- `hpx::partition_copy`
- `hpx::remove`
- `hpx::remove_copy`
- `hpx::remove_copy_if`
- `hpx::remove_if`
- `hpx::replace`
- `hpx::replace_copy`
- `hpx::replace_copy_if`
- `hpx::replace_if`
- `hpx::reverse`
- `hpx::reverse_copy`
- `hpx::rotate`
- `hpx::rotate_copy`
- `hpx::search`
- `hpx::search_n`
- `hpx::set_difference`
- `hpx::set_intersection`
- `hpx::set_symmetric_difference`
- `hpx::set_union`
- `hpx::shift_left`
- `hpx::shift_right`
- `hpx::sort`
- `hpx::stable_partition`
- `hpx::stable_sort`
- `hpx::starts_with`
- `hpx::swap_ranges`
- `hpx::transform`
- `hpx::unique`
- `hpx::unique_copy`
- `hpx::for_loop`
- `hpx::for_loop_strided`
- `hpx::for_loop_n`
- `hpx::for_loop_n_strided`
- `hpx::ranges::adjacent_find`
- `hpx::ranges::all_of`
- `hpx::ranges::any_of`

- `hpx::ranges::copy`
- `hpx::ranges::copy_if`
- `hpx::ranges::copy_n`
- `hpx::ranges::count`
- `hpx::ranges::count_if`
- `hpx::ranges::ends_with`
- `hpx::ranges::equal`
- `hpx::ranges::fill`
- `hpx::ranges::fill_n`
- `hpx::ranges::find`
- `hpx::ranges::find_end`
- `hpx::ranges::find_first_of`
- `hpx::ranges::find_if`
- `hpx::ranges::find_if_not`
- `hpx::ranges::for_each`
- `hpx::ranges::for_each_n`
- `hpx::ranges::generate`
- `hpx::ranges::generate_n`
- `hpx::ranges::includes`
- `hpx::ranges::inplace_merge`
- `hpx::ranges::is_heap`
- `hpx::ranges::is_heap_until`
- `hpx::ranges::is_partitioned`
- `hpx::ranges::is_sorted`
- `hpx::ranges::is_sorted_until`
- `hpx::ranges::make_heap`
- `hpx::ranges::merge`
- `hpx::ranges::move`
- `hpx::ranges::none_of`
- `hpx::ranges::nth_element`
- `hpx::ranges::partial_sort`
- `hpx::ranges::partial_sort_copy`
- `hpx::ranges::partition`
- `hpx::ranges::partition_copy`
- `hpx::ranges::set_difference`
- `hpx::ranges::set_intersection`

- `hpx::ranges::set_symmetric_difference`
- `hpx::ranges::set_union`
- `hpx::ranges::shift_left`
- `hpx::ranges::shift_right`
- `hpx::ranges::sort`
- `hpx::ranges::stable_partition`
- `hpx::ranges::stable_sort`
- `hpx::ranges::starts_with`
- `hpx::ranges::swap_ranges`
- `hpx::ranges::unique`
- `hpx::ranges::unique_copy`
- `hpx::ranges::for_loop`
- `hpx::ranges::for_loop_strided`

Header `hpx/any.hpp`

This header includes *Header hpx/local/any.hpp*.

Header `hpx/local/any.hpp`

Corresponds to the C++ standard library header `any`¹⁹³. `hpx::any` is compatible with `std::any`.

Classes

- `hpx::any`
- `hpx::any_nonser`
- `hpx::bad_any_cast`
- `hpx::unique_any_nonser`

Functions

- `hpx::any_cast`
- `hpx::make_any`
- `hpx::make_any_nonser`
- `hpx::make_unique_any_nonser`

¹⁹³ <http://en.cppreference.com/w/cpp/header/any>

Header `hpx/assert.hpp`

Corresponds to the C++ standard library header `cassert`¹⁹⁴. `HPX_ASSERT` is the HPX equivalent to `assert` in `cassert`. `HPX_ASSERT` can also be used in CUDA device code.

Macros

- `HPX_ASSERT`
- `HPX_ASSERT_MSG`

Header `hpx/barrier.hpp`

This header includes `Header hpx/local/barrier.hpp` and contains a distributed barrier implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

- `hpx::lcos::barrier`

Header `hpx/local/barrier.hpp`

Corresponds to the C++ standard library header `barrier`¹⁹⁵.

Classes

- `hpx::lcos::local::cpp20_barrier`

Header `hpx/channel.hpp`

This header includes `Header hpx/local/channel.hpp` and contains a distributed channel implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

- `hpx::lcos::channel`

¹⁹⁴ <http://en.cppreference.com/w/cpp/header/cassert>

¹⁹⁵ <http://en.cppreference.com/w/cpp/header/barrier>

Header `hpx/local/channel.hpp`

Contains a local channel implementation.

Classes

- `hpx::lcos::local::channel`

Header `hpx/chrono.hpp`

This header includes *Header hpx/local/chrono.hpp*.

Header `hpx/local/chrono.hpp`

Corresponds to the C++ standard library header `chrono`¹⁹⁶. The following replacements and extensions are provided compared to `chrono`¹⁹⁷. The classes below are also available in the `hpx::chrono` namespace, not in the top-level `hpx` namespace.

Classes

- `hpx::chrono::high_resolution_clock`
- `hpx::chrono::high_resolution_timer`
- `hpx::chrono::steady_time_point`

Header `hpx/condition_variable.hpp`

This header includes *Header hpx/local/condition_variable.hpp*.

Header `hpx/local/condition_variable.hpp`

Corresponds to the C++ standard library header `condition_variable`¹⁹⁸.

Classes

- `hpx::lcos::local::condition_variable`
- `hpx::lcos::local::condition_variable_any`
- `hpx::lcos::local::cv_status`

¹⁹⁶ <http://en.cppreference.com/w/cpp/header/chrono>

¹⁹⁷ <http://en.cppreference.com/w/cpp/header/chrono>

¹⁹⁸ http://en.cppreference.com/w/cpp/header/condition_variable

Header `hpx/exception.hpp`

This header includes *Header hpx/local/exception.hpp*.

Header `hpx/local/exception.hpp`

Corresponds to the C++ standard library header `exception`¹⁹⁹. `hpx::exception` extends `std::exception` and is the base class for all exceptions thrown in HPX. `HPX_THROW_EXCEPTION` can be used to throw HPX exceptions with file and line information attached to the exception.

Macros

- `HPX_THROW_EXCEPTION`

Classes

- `hpx::exception`

Header `hpx/execution.hpp`

This header includes *Header hpx/local/execution.hpp*.

Header `hpx/local/execution.hpp`

Corresponds to the C++ standard library header `execution`²⁰⁰. See *High level parallel facilities, Using parallel algorithms* and *Executor parameters and executor parameter traits* for more information about execution policies and executor parameters.

Note: These names are only available in the `hpx::execution` namespace, not in the top-level `hpx` namespace.

Constants

- `hpx::execution::seq`
- `hpx::execution::par`
- `hpx::execution::par_unseq`
- `hpx::execution::task`

¹⁹⁹ <http://en.cppreference.com/w/cpp/header/exception>

²⁰⁰ <http://en.cppreference.com/w/cpp/header/execution>

Classes

- `hpx::execution::sequenced_policy`
- `hpx::execution::parallel_policy`
- `hpx::execution::parallel_unsequenced_policy`
- `hpx::execution::sequenced_task_policy`
- `hpx::execution::parallel_task_policy`
- `hpx::execution::auto_chunk_size`
- `hpx::execution::dynamic_chunk_size`
- `hpx::execution::guided_chunk_size`
- `hpx::execution::persistent_auto_chunk_size`
- `hpx::execution::static_chunk_size`

Header `hpx/functional.hpp`

This header includes *Header hpx/local/functional.hpp*.

Header `hpx/local/functional.hpp`

Corresponds to the C++ standard library header `functional`²⁰¹. `hpx::util::function` is a more efficient and serializable replacement for `std::function`.

Constants

The following constants are also available in `hpx::placeholders`, not the top-level `hpx` namespace.

- `hpx::util::placeholders::_1`
- `hpx::util::placeholders::_2`
- `...`
- `hpx::util::placeholders::_9`

Classes

- `hpx::util::function`
- `hpx::util::function_nonser`
- `hpx::util::function_ref`
- `hpx::util::unique_function`
- `hpx::util::unique_function_nonser`
- `hpx::traits::is_bind_expression`
- `hpx::traits::is_placeholder`

²⁰¹ <http://en.cppreference.com/w/cpp/header/functional>

- `hpx::scoped_annotation`

Functions

- `hpx::annotated_function`
- `hpx::util::bind`
- `hpx::util::bind_back`
- `hpx::util::bind_front`
- `hpx::util::invoke`
- `hpx::util::invoke_fused`
- `hpx::util::mem_fn`

Header `hpx/future.hpp`

This header includes *Header hpx/local/future.hpp* and contains overloads of `hpx::async`, `hpx::apply`, `hpx::sync`, and `hpx::dataflow` that can be used with actions. See *Action invocation* for more information about invoking actions.

Note: The alias from `hpx::promise` to `hpx::lcos::promise` is deprecated and will be removed in a future release. The alias `hpx::distributed::promise` should be used in new applications.

Classes

- `hpx::lcos::promise`

Functions

- `hpx::async`
- `hpx::apply`
- `hpx::sync`
- `hpx::dataflow`

Header `hpx/local/future.hpp`

Corresponds to the C++ standard library header `future`²⁰². See *Extended facilities for futures* for more information about extensions to futures compared to the C++ standard library.

Note: All names except `hpx::lcos::local::promise` are also available in the top-level `hpx` namespace. `hpx::promise` refers to `hpx::lcos::promise`, a distributed variant of `hpx::lcos::local::promise`, but will eventually refer to `hpx::lcos::local::promise` after a deprecation period.

²⁰² <http://en.cppreference.com/w/cpp/header/future>

Classes

- `hpx::future`
- `hpx::shared_future`
- `hpx::lcos::local::promise`
- `hpx::launch`

Functions

- `hpx::make_future`
- `hpx::make_shared_future`
- `hpx::make_ready_future`
- `hpx::make_ready_future_alloc`
- `hpx::make_ready_future_at`
- `hpx::make_ready_future_after`
- `hpx::make_exceptional_future`
- `hpx::async`
- `hpx::apply`
- `hpx::sync`
- `hpx::dataflow`
- `hpx::when_all`
- `hpx::when_any`
- `hpx::when_some`
- `hpx::when_each`
- `hpx::wait_all`
- `hpx::wait_any`
- `hpx::wait_some`
- `hpx::wait_each`

Examples

```
#include <hpx/assert.hpp>
#include <hpx/future.hpp>
#include <hpx/hpx_main.hpp>
#include <hpx/tuple.hpp>

#include <iostream>
#include <utility>

int main()
{
```

(continues on next page)

(continued from previous page)

```

// Asynchronous execution with futures
hpx::future<void> f1 = hpx::async(hpx::launch::async, []() {});
hpx::shared_future<int> f2 =
    hpx::async(hpx::launch::async, []() { return 42; });
hpx::future<int> f3 =
    f2.then([](hpx::shared_future<int>&& f) { return f.get() * 3; });

hpx::lcos::local::promise<double> p;
auto f4 = p.get_future();
HPX_ASSERT(!f4.is_ready());
p.set_value(123.45);
HPX_ASSERT(f4.is_ready());

hpx::packaged_task<int()> t([]() { return 43; });
hpx::future<int> f5 = t.get_future();
HPX_ASSERT(!f5.is_ready());
t();
HPX_ASSERT(f5.is_ready());

// Fire-and-forget
hpx::apply([]() {
    std::cout << "This will be printed later\n" << std::flush;
});

// Synchronous execution
hpx::sync([]() {
    std::cout << "This will be printed immediately\n" << std::flush;
});

// Combinators
hpx::future<double> f6 = hpx::async([]() { return 3.14; });
hpx::future<double> f7 = hpx::async([]() { return 42.0; });
std::cout
    << hpx::when_all(f6, f7)
        .then([](hpx::future<
                    hpx::tuple<hpx::future<double>, hpx::future<double>>>
            f) {
            hpx::tuple<hpx::future<double>, hpx::future<double>> t =
                f.get();
            double pi = hpx::get<0>(t).get();
            double r = hpx::get<1>(t).get();
            return pi * r * r;
        })
        .get()
    << std::endl;

// Easier continuations with dataflow; it waits for all future or
// shared_future arguments before executing the continuation, and also
// accepts non-future arguments
hpx::future<double> f8 = hpx::async([]() { return 3.14; });
hpx::future<double> f9 = hpx::make_ready_future(42.0);
hpx::shared_future<double> f10 = hpx::async([]() { return 123.45; });
hpx::future<hpx::tuple<double, double>> f11 = hpx::dataflow(
    [] (hpx::future<double> a, hpx::future<double> b,
        hpx::shared_future<double> c, double d) {
        return hpx::make_tuple<>(a.get() + b.get(), c.get() / d);
},

```

(continues on next page)

(continued from previous page)

```

f8, f9, f10, -3.9);

// split_future gives a tuple of futures from a future of tuple
hpx::tuple<hpx::future<double>, hpx::future<double>> f12 =
    hpx::split_future(std::move(f11));
std::cout << hpx::get<1>(f12).get() << std::endl;

return 0;
}

```

Header `hpx/init.hpp`

This header contains functionality for starting, stopping, suspending, and resuming the *HPX* runtime. This is the main way to explicitly start the *HPX* runtime. See *Starting the HPX runtime* for more details on starting the *HPX* runtime.

Classes

- `hpx::init_params`
- `hpx::runtime_mode`

Functions

- `hpx::init`
- `hpx::start`
- `hpx::finalize`
- `hpx::disconnect`
- `hpx::suspend`
- `hpx::resume`

Header `hpx/latch.hpp`

This header includes *Header hpx/local/latch.hpp* and contains a distributed latch implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

- `hpx::lcos::latch`

Header `hpx/local/latch.hpp`

Corresponds to the C++ standard library header `latch`²⁰³.

Classes

- `hpx::lcos::local::cpp20_latch`

Header `hpx/local/mutex.hpp`

This header includes *Header hpx/local/mutex.hpp*.

Header `hpx/local/mutex.hpp`

Corresponds to the C++ standard library header `mutex`²⁰⁴.

Classes

- `hpx::lcos::local::mutex`
- `hpx::lcos::local::no_mutex`
- `hpx::lcos::local::once_flag`
- `hpx::lcos::local::recursive_mutex`
- `hpx::lcos::local::spinlock`
- `hpx::lcos::local::timed_mutex`
- `hpx::lcos::local::unlock_guard`

Functions

- `hpx::lcos::local::call_once`

Header `hpx/memory.hpp`

This header includes *Header hpx/local/memory.hpp*.

²⁰³ <http://en.cppreference.com/w/cpp/header/latch>

²⁰⁴ <http://en.cppreference.com/w/cpp/header/mutex>

Header `hpx/local/memory.hpp`

Corresponds to the C++ standard library header `memory`²⁰⁵. It contains parallel versions of the copy, fill, move, and construct helper functions in `memory`²⁰⁶. See *Using parallel algorithms* for more information about the parallel algorithms.

Functions

- `hpx::uninitialized_copy`
- `hpx::uninitialized_copy_n`
- `hpx::uninitialized_default_construct`
- `hpx::uninitialized_default_construct_n`
- `hpx::uninitialized_fill`
- `hpx::uninitialized_fill_n`
- `hpx::uninitialized_move`
- `hpx::uninitialized_move_n`
- `hpx::uninitialized_value_construct`
- `hpx::uninitialized_value_construct_n`
- `hpx::ranges::uninitialized_copy`
- `hpx::ranges::uninitialized_copy_n`
- `hpx::ranges::uninitialized_default_construct`
- `hpx::ranges::uninitialized_default_construct_n`
- `hpx::ranges::uninitialized_fill`
- `hpx::ranges::uninitialized_fill_n`
- `hpx::ranges::uninitialized_move`
- `hpx::ranges::uninitialized_move_n`
- `hpx::ranges::uninitialized_value_construct`
- `hpx::ranges::uninitialized_value_construct_n`

Header `hpx/numeric.hpp`

This header includes *Header `hpx/local/numeric.hpp`*.

²⁰⁵ <http://en.cppreference.com/w/cpp/header/memory>

²⁰⁶ <http://en.cppreference.com/w/cpp/header/memory>

Header `hpx/local/numeric.hpp`

Corresponds to the C++ standard library header `numeric`²⁰⁷. See *Using parallel algorithms* for more information about the parallel algorithms.

Functions

- `hpx::adjacent_difference`
- `hpx::exclusive_scan`
- `hpx::inclusive_scan`
- `hpx::reduce`
- `hpx::transform_exclusive_scan`
- `hpx::transform_inclusive_scan`
- `hpx::transform_reduce`
- `hpx::ranges::exclusive_scan`
- `hpx::ranges::inclusive_scan`
- `hpx::ranges::transform_exclusive_scan`
- `hpx::ranges::transform_inclusive_scan`

Header `hpx/optional.hpp`

This header includes *Header hpx/local/optional.hpp*.

Header `hpx/local/optional.hpp`

Corresponds to the C++ standard library header `optional`²⁰⁸. `hpx::util::optional` is compatible with `std::optional`.

Constants

- `hpx::util::nullopt`

Classes

- `hpx::util::optional`
- `hpx::util::nullopt_t`
- `hpx::util::bad_optional_access`

²⁰⁷ <http://en.cppreference.com/w/cpp/header/numeric>

²⁰⁸ <http://en.cppreference.com/w/cpp/header/optional>

Functions

- `hpx::util::make_optional`

Header `hpx/runtime.hpp`

This header includes *Header hpx/local/runtime.hpp* and contains functions for accessing distributed runtime information.

Functions

- `hpx::find_root_locality`
- `hpx::find_all_localities`
- `hpx::find_remote_localities`
- `hpx::find_locality`
- `hpx::get_colocation_id`
- `hpx::get_locality_id`

Header `hpx/local/runtime.hpp`

This header contains functions for accessing local runtime information.

Typedefs

- `hpx::startup_function_type`
- `hpx::shutdown_function_type`

Functions

- `hpx::get_num_worker_threads`
- `hpx::get_worker_thread_num`
- `hpx::get_thread_name`
- `hpx::register_pre_startup_function`
- `hpx::register_startup_function`
- `hpx::register_pre_shutdown_function`
- `hpx::register_shutdown_function`
- `hpx::get_num_localities`
- `hpx::get_locality_name`

Header `hpx/system_error.hpp`

This header includes *Header hpx/local/system_error.hpp*.

Header `hpx/local/system_error.hpp`

Corresponds to the C++ standard library header `system_error`²⁰⁹.

Classes

- `hpx::error_code`

Header `hpx/task_block.hpp`

This header includes *Header hpx/local/task_black.hpp*.

Header `hpx/local/task_black.hpp`

Corresponds to the `task_block` feature in N4411²¹⁰. See *Using task blocks* for more details on using task blocks.

Classes

- `hpx::parallel::v2::task_canceled_exception`
- `hpx::parallel::v2::task_block`

Functions

- `hpx::parallel::v2::define_task_block`
- `hpx::parallel::v2::define_task_block_restore_thread`

Header `hpx/thread.hpp`

This header includes *Header hpx/local/thread.hpp*.

Header `hpx/local/thread.hpp`

Corresponds to the C++ standard library header `thread`²¹¹. The functionality in this header is equivalent to the standard library thread functionality, with the exception that the *HPX* equivalents are implemented on top of lightweight threads and the *HPX* runtime.

²⁰⁹ http://en.cppreference.com/w/cpp/header/system_error

²¹⁰ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

²¹¹ <http://en.cppreference.com/w/cpp/header/thread>

Classes

- `hpx::thread`
- `hpx::jthread`

Functions

- `hpx::this_thread::yield`
- `hpx::this_thread::get_id`
- `hpx::this_thread::sleep_for`
- `hpx::this_thread::sleep_until`

Header `hpx/semaphore.hpp`

This header includes *Header hpx/local/semaphore.hpp*.

Header `hpx/local/semaphore.hpp`

Corresponds to the C++ standard library header `semaphore`²¹².

Classes

- `hpx::lcos::local::cpp20_binary_semaphore`
- `hpx::lcos::local::cpp20_counting_semaphore`

Header `hpx/shared_mutex.hpp`

This header includes *Header hpx/local/shared_mutex.hpp*.

Header `hpx/local/shared_mutex.hpp`

Corresponds to the C++ standard library header `shared_mutex`²¹³.

Classes

- `hpx::lcos::local::shared_mutex`

²¹² <http://en.cppreference.com/w/cpp/header/semaphore>

²¹³ http://en.cppreference.com/w/cpp/header/shared_mutex

Header `hpx/stop_token.hpp`

This header includes *Header hpx/local/stop_token.hpp*.

Header `hpx/local/stop_token.hpp`

Corresponds to the C++ standard library header `stop_token`²¹⁴.

Constants

- `hpx::nostopstate`

Classes

- `hpx::stop_callback`
- `hpx::stop_source`
- `hpx::stop_token`
- `hpx::nostopstate_t`

Header `hpx/tuple.hpp`

This header includes *Header hpx/local/tuple.hpp*.

Header `hpx/local/tuple.hpp`

Corresponds to the C++ standard library header `tuple`²¹⁵. `hpx::tuple` can be used in CUDA device code, unlike `std::tuple`.

Constants

- `hpx::ignore`

Classes

- `hpx::tuple`
- `hpx::tuple_size`
- `hpx::tuple_element`

²¹⁴ http://en.cppreference.com/w/cpp/header/stop_token

²¹⁵ <http://en.cppreference.com/w/cpp/header/tuple>

Functions

- `hpx::make_tuple`
- `hpx::tie`
- `hpx::forward_as_tuple`
- `hpx::tuple_cat`
- `hpx::get`

Header `hpx/type_traits.hpp`

This header includes *Header hpx/local/type_traits.hpp*.

Header `hpx/local/type_traits.hpp`

Corresponds to the C++ standard library header `type_traits`²¹⁶.

Classes

- `hpx::is_invocable`
- `hpx::is_invocable_r`

Header `hpx/unwrap.hpp`

This header includes *Header hpx/local/unwrap.hpp*.

Header `hpx/local/unwrap.hpp`

Contains utilities for unwrapping futures.

Classes

- `hpx::functional::unwrap`
- `hpx::functional::unwrap_n`
- `hpx::functional::unwrap_all`

²¹⁶ http://en.cppreference.com/w/cpp/header/type_traits

Functions

- `hpx::unwrap`
- `hpx::unwrap_n`
- `hpx::unwrap_all`
- `hpx::unwrapping`
- `hpx::unwrapping_n`
- `hpx::unwrapping_all`

Header `hpx/version.hpp`

This header provides version information about *HPX*.

Macros

- `HPX_VERSION_MAJOR`
- `HPX_VERSION_MINOR`
- `HPX_VERSION_SUBMINOR`
- `HPX_VERSION_FULL`
- `HPX_VERSION_DATE`
- `HPX_VERSION_TAG`
- `HPX_AGAS_VERSION`

Functions

- `hpx::major_version`
- `hpx::minor_version`
- `hpx::subminor_version`
- `hpx::full_version`
- `hpx::full_version_as_string`
- `hpx::tag`
- `hpx::agas_version`
- `hpx::build_type`
- `hpx::build_date_time`

Header `hpx/wrap_main.hpp`

This header does not provide any direct functionality but is used for implicitly using `main` as the runtime entry point. See *Re-use the `main()` function as the main HPX entry point* for more details on implicitly starting the *HPX* runtime.

2.8.2 Full API

The full API of *HPX* is presented below. The listings for the public API above refer to the full documentation below.

Note: Most names listed in the full API reference are implementation details or considered unstable. They are listed mostly for completeness. If there is a particular feature you think deserves being in the public API we may consider promoting it. In general we prioritize making sure features corresponding to C++ standard library features are stable and complete.

Main *HPX* library

This lists functionality in the main *HPX* library that has not been moved to modules yet.

`namespace hpx`

`namespace components`

Functions

```
template<typename Component>
future<naming::id_type> migrate_from_storage(naming::id_type const &to_resurrect,
                                                 naming::id_type const &target = naming::invalid_id)
```

Migrate the component with the given id from the specified target storage (resurrect the object)

The function *migrate_from_storage*<*Component*> will migrate the component referenced by *to_resurrect* from the storage facility specified where the object is currently stored on. It returns a future referring to the migrated component instance. The component instance is resurrected on the locality specified by *target_locality*.

Return A future representing the global id of the migrated component instance. This should be the same as *to_resurrect*.

Parameters

- *to_resurrect*: [in] The global id of the component to migrate.
- *target*: [in] The optional locality to resurrect the object on. By default the object is resurrected on the locality it was located on last.

Template Parameters

- The: only template argument specifies the component type of the component to migrate from the given storage facility.

`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

- *Type*: only template argument specifies the component type of the component to migrate to the given storage facility.

```
template<typename Derived, typename StubDerived 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.

file migrate_from_storage.hpp

```
#include <hpx/config.hpp>#include <hpx/components_base/traits/is_component.hpp>#include
<hpx/futures/future.hpp>#include <hpx/naming_base/id_type.hpp>#include <hpx/components/component_storage/server/migrat
<type_traits>
```

file migrate_to_storage.hpp

```
#include <hpx/config.hpp>#include <hpx/components/client_base.hpp>#include
<hpx/components_base/traits/is_component.hpp>#include <hpx/futures/future.hpp>#include
<hpx/naming_base/id_type.hpp>#include <hpx/components/component_storage/component_storage.hpp>#include
<hpx/components/component_storage/server/migrate_to_storage.hpp>#include <type_traits>
```

dir /hpx/source/components/component_storage

dir /hpx/source/components/component_storage/include/hpx/components/component_storage

dir /hpx/source/components/component_storage/include/hpx/components

dir /hpx/source/components

dir /hpx/source/components/component_storage/include/hpx

dir /hpx/source/components/component_storage/include

`dir /hpx/source`

affinity

See *Public API* for a list of names and headers that are part of the public HPX API.

`namespace hpx`

`namespace threads`

Functions

```
void parse_affinity_options(std::string const &spec, std::vector<mask_type> &affinities, std::size_t used_cores, std::size_t max_cores, std::size_t num_threads, std::vector<std::size_t> &num_pus, bool use_process_mask, error_code &ec = throws)
```

```
void parse_affinity_options(std::string const &spec, std::vector<mask_type> &affinities, error_code &ec = throws)
```

algorithms

See *Public API* for a list of names and headers that are part of the public HPX API.

`namespace hpx`

`namespace traits`

Typedefs

```
template<typename T>
using proxy_value_t = typename proxy_value<T>::type
```

Variables

```
template<typename T>
constexpr bool is_value_proxy_v = is_value_proxy<T>::value
```

```
template<typename T>
struct proxy_value
```

Public Types

```
template<>
using type = T

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename Source, typename Dest>
using pointer_copy_category_t = typename pointer_copy_category<Source, Dest>::type

template<typename Source, typename Dest>
using pointer_move_category_t = typename pointer_move_category<Source, Dest>::type

template<typename Iterator>
using remove_const_iterator_value_type_t = typename remove_const_iterator_value_type<Iterator>::type

struct general_pointer_tag
    Subclassed by hpx::traits::trivially_copyable_pointer_tag

template<typename Source, typename Dest, typename Enable = void>
struct pointer_copy_category
```

Public Types

```
template<>
using type = typename detail::pointer_copy_category::type

template<typename Source, typename Dest, typename Enable = void>
struct pointer_move_category
```

Public Types

```
template<>
using type = typename detail::pointer_move_category::type

template<typename Iterator, typename Enable = void>
struct remove_const_iterator_value_type
```

Public Types

```
template<>
using type = Iterator

template<typename Iterator>
struct projected_iterator<Iterator, typename std::enable_if<is_segmented_iterator<Iterator>::value>::type>
```

Public Types

```
template<>
using local_iterator = typename segmented_iterator_traits::local_iterator

template<>
using type = typename segmented_local_iterator_traits<local_iterator>::local_raw_iterator

template<typename Iterator>
struct projected_iterator<Iterator, typename hpx::util::always_void<typename std::decay<Iterator>::type::proxy_type>
```

Public Types

```
template<>
using type = typename std::decay<Iterator>::type::proxy_type

namespace hpx

namespace parallel

namespace traits
```

Typedefs

```
template<typename F, typename Iter>
using is_projected_t = typename is_projected<F, Iter>::type

template<typename ExPolicy, typename F, typename ...Projected>
using is_indirect_callable_t = typename is_indirect_callable<ExPolicy, F, Projected...>::type
```

Variables

```
template<typename F, typename Iter>
constexpr bool is_projected_v = is_projected<F, Iter>::value

template<typename ExPolicy, typename F, typename ...Projected>
constexpr bool is_indirect_callable_v = is_indirect_callable<ExPolicy, F, Projected...>::value

template<typename Proj, typename Iter>
struct projected
```

Public Types

```
template<>
using projector_type = typename std::decay<Proj>::type

template<>
using iterator_type = typename hpx::traits::projected_iterator<Iter>::type

namespace traits

template<typename T, typename Enable = void>
struct projected_iterator
```

Public Types

```
template<>
using type = typename std::decay::type

template<typename Iterator>
struct projected_iterator<Iterator, typename hpx::util::always_void<typename std::decay<Iterator>::type>
```

Public Types

```
template<>
using type = typename std::decay<Iterator>::type::proxy_type

template<typename Iterator>
struct projected_iterator<Iterator, typename std::enable_if<is_segmented_iterator<Iterator>::value>::type>
```

Public Types

```
template<>
using local_iterator = typename segmented_iterator_traits::local_iterator

template<>
using type = typename segmented_local_iterator_traits<local_iterator>::local_raw_iterator

template<typename Proj, typename Rng>
struct projected_range<Proj, Rng, typename std::enable_if<hpx::traits::is_range<Rng>::value>::type>
```

Public Types

```
template<>
using projector_type = typename std::decay<Proj>::type

template<>
using iterator_type = typename hpx::traits::range_iterator<Rng>::type
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace traits
```

Variables

```
template<typename Proj, typename Rng>
constexpr bool is_projected_range_v = is_projected_range<Proj, Rng>::value

template<typename Proj, typename Rng>
struct projected_range<Proj, Rng, typename std::enable_if<hpx::traits::is_range<Rng>::value>::type>
```

Public Types

```
template<>
using projector_type = typename std::decay<Proj>::type

template<>
using iterator_type = typename hpx::traits::range_iterator<Rng>::type

namespace hpx

namespace traits

template<typename Iterator, typename Enable = void>
struct segmented_iterator_traits
```

Public Types

```
typedef std::false_type is_segmented_iterator

template<typename Iterator, typename Enable = void>
struct segmented_local_iterator_traits
```

Public Types

```
typedef std::false_type is_segmented_local_iterator

typedef Iterator iterator
typedef Iterator local_iterator
typedef Iterator local_raw_iterator
```

Public Static Functions

```
static local_raw_iterator const &local (local_iterator const &it)

static local_iterator const &remote (local_raw_iterator const &it)

static local_raw_iterator local (local_iterator &&it)

static local_iterator remote (local_raw_iterator &&it)

namespace hpx
```

```
namespace lcos
```

```
namespace local
```

Functions

```
template<typename ExPolicy, typename F, typename ...Args, typename = typename std::enable_if<hpx::parallel::is_parallel_fn<F>>
define_spmd_block(ExPolicy &&policy, std::size_t
num_images, F &&f, Args&&...
args)
```

```
template<typename ExPolicy, typename F, typename ...Args, typename = typename std::enable_if<!hpx::is_asynchronous_fn<F>>
void define_spmd_block(ExPolicy &&policy, std::size_t num_images, F &&f, Args&&...
args)
```

```
template<typename F, typename ...Args>
void define_spmd_block(std::size_t num_images, F &&f, Args&&... args)
```

```
struct spmd_block
```

#include <spmd_block.hpp> The class *spmd_block* defines an interface for launching multiple images while giving handles to each image to interact with the remaining images. The *define_spmd_block* function templates create multiple images of a user-defined function (or lambda) and launches them in a possibly separate thread. A temporary spmd block object is created and diffused to each image. The constraint for the function (or lambda) given to the *define_spmd_block* function is to accept a *spmd_block* as first parameter.

Public Functions

```
spmd_block(std::size_t num_images, std::size_t image_id, barrier_type &barrier, table_type &barriers, mutex_type &mtx)
```

```
spmd_block(spmd_block&&)
```

```
spmd_block(spmd_block const&)
```

```
spmd_block &operator=(spmd_block&&)
```

```
spmd_block &operator=(spmd_block const&)
```

```
std::size_t get_num_images() const
```

```
std::size_t this_image() const
```

```
void sync_all() const
```

```
void sync_images(std::set<std::size_t> const &images) const
```

```
void sync_images(std::vector<std::size_t> const &input_images) const
```

```
template<typename Iterator>
std::enable_if<traits::is_input_iterator<Iterator>::value>::type sync_images(Iterator
begin, Iterator end)
const
```

```
template<typename ...I>
std::enable_if<util::all_of<typename std::is_integral<I>::value...>::value>::type sync_images(I...
i)
const
```

Private Types

```
using barrier_type = hpx::lcos::local::barrier
using table_type = std::map<std::set<std::size_t>, std::shared_ptr<barrier_type>>
using mutex_type = hpx::lcos::local::mutex
```

Private Members

```
std::size_t num_images_
std::size_t image_id_
std::reference_wrapper<barrier_type> barrier_
std::reference_wrapper<table_type> barriers_
std::reference_wrapper<mutex_type> mtx_
```

```
namespace parallel
```

Typedefs

```
using spmd_block = hpx::lcos::local::spmd_block
```

The class `spmd_block` defines an interface for launching multiple images while giving handles to each image to interact with the remaining images. The `define_spmd_block` function templates create multiple images of a user-defined function (or lambda) and launches them in a possibly separate thread. A temporary `spmd_block` object is created and diffused to each image. The constraint for the function (or lambda) given to the `define_spmd_block` function is to accept a `spmd_block` as first parameter.

Functions

```
template<typename ExPolicy, typename F, typename ...Args, typename = typename std::enable_if<hpx::parallel::executors::is_parallel_executor_v<ExPolicy>> define_spmd_block(ExPolicy &&policy, std::size_t num_images, F &&f, Args&&... args)
```

```
template<typename ExPolicy, typename F, typename ...Args, typename = typename std::enable_if<!hpx::is_async_executing_v<ExPolicy>> define_spmd_block(ExPolicy &&policy, std::size_t num_images, F &&f, Args&&... args)
```

```
template<typename F, typename ...Args>
void define_spmd_block(std::size_t num_images, F &&f, Args&&... args)
```

```
namespace hpx
```

```
namespace parallel
```

Functions

```
template<typename ExPolicy, typename F>
hpx::future<void> define_task_block(ExPolicy &&policy, F &&f)
Constructs a task_block, tr, using the given execution policy policy, and invokes the expression f(tr) on the user-provided object, f.
```

Postcondition: All tasks spawned from *f* have finished execution. A call to `define_task_block` may return on a different thread than that on which it was called.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the task block may be parallelized.
- *F*: The type of the user defined function to invoke inside the `define_task_block` (deduced). *F* shall be `MoveConstructible`.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *f*: The user defined function to invoke inside the task block. Given an lvalue *tr* of type `task_block`, the expression, `(void)f(tr)`, shall be well-formed.

Note It is expected (but not mandated) that *f* will (directly or indirectly) call `tr.run(callable_object)`.

Exceptions

- An: *exception_list*, as specified in Exception Handling.

```
template<typename ExPolicy, typename F>
void define_task_block(ExPolicy &&policy, F &&f)
```

```
template<typename F>
void define_task_block(F &&f)
```

Constructs a task_block, *tr*, and invokes the expression *f(tr)* on the user-provided object, *f*. This version uses `parallel_policy` for task scheduling.

Postcondition: All tasks spawned from *f* have finished execution. A call to `define_task_block` may return on a different thread than that on which it was called.

Template Parameters

- *F*: The type of the user defined function to invoke inside the `define_task_block` (deduced). *F* shall be `MoveConstructible`.

Parameters

- *f*: The user defined function to invoke inside the task block. Given an lvalue *tr* of type `task_block`, the expression, `(void)f(tr)`, shall be well-formed.

Note It is expected (but not mandated) that *f* will (directly or indirectly) call `tr.run(callable_object)`.

Exceptions

- An: *exception_list*, as specified in Exception Handling.

```
template<typename ExPolicy, typename F>
util::detail::algorithm_result<ExPolicy>::type define_task_block_restore_thread(ExPolicy &&policy, F &&f)
```

Constructs a task_block, *tr*, and invokes the expression *f(tr)* on the user-provided object, *f*.

Postcondition: All tasks spawned from f have finished execution. A call to `define_task_block_restore_thread` always returns on the same thread as that on which it was called.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the task block may be parallelized.
- F : The type of the user defined function to invoke inside the `define_task_block` (deduced). F shall be `MoveConstructible`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- f : The user defined function to invoke inside the `define_task_block`. Given an lvalue tr of type `task_block`, the expression, `(void)f(tr)`, shall be well-formed.

Exceptions

- An: `exception_list`, as specified in Exception Handling.

Note It is expected (but not mandated) that f will (directly or indirectly) call `tr.run(callable_object)`.

```
template<typename F>
void define_task_block_restore_thread(F &&f)
Constructs a task_block, tr, and invokes the expression f(tr) on the user-provided object, f. This version uses parallel_policy for task scheduling.
```

Postcondition: All tasks spawned from f have finished execution. A call to `define_task_block_restore_thread` always returns on the same thread as that on which it was called.

Template Parameters

- F : The type of the user defined function to invoke inside the `define_task_block` (deduced). F shall be `MoveConstructible`.

Parameters

- f : The user defined function to invoke inside the `define_task_block`. Given an lvalue tr of type `task_block`, the expression, `(void)f(tr)`, shall be well-formed.

Exceptions

- An: `exception_list`, as specified in Exception Handling.

Note It is expected (but not mandated) that f will (directly or indirectly) call `tr.run(callable_object)`.

```
namespace v2
```

```
template<typename ExPolicy = hpx::execution::parallel_policy>
class task_block
#include <task_block.hpp> The class task_block defines an interface for forking and joining parallel tasks. The define_task_block and define_task_block_restore_thread function templates create an object of type task_block and pass a reference to that object to a user-provided callable object.
```

An object of class `task_block` cannot be constructed, destroyed, copied, or moved except by the implementation of the task region library. Taking the address of a `task_block` object via operator& or addressof is ill formed. The result of obtaining its address by any other means is unspecified.

A `task_block` is active if it was created by the nearest enclosing task block, where “task block” refers to an invocation of `define_task_block` or `define_task_block_restore_thread` and “nearest enclosing” means the most recent invocation that has not yet completed. Code designated for execution in another thread by means other than the facilities in this section (e.g., using `thread` or `async`) are not enclosed in the task region and a `task_block` passed to (or captured by) such code

is not active within that code. Performing any operation on a *task_block* that is not active results in undefined behavior.

The *task_block* that is active before a specific call to the *run* member function is not active within the asynchronous function that invoked *run*. (The invoked function should not, therefore, capture the *task_block* from the surrounding block.)

Example:

```
define_task_block([&] (auto& tr) {
    tr.run([&] {
        tr.run([] { f(); });
    });
    // Error: tr is not active
    define_task_block([&] (auto& tr) {
        tr.run(f());
    });
    // Nested task block
    // 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

```
template<>
using execution_policy = ExPolicy
```

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. The function will also throw a *exception_list* holding all exceptions that were caught while executing the tasks.

```
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.
The function will also throw a `exception_list` holding all exceptions that were caught while executing the tasks.

`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

`hpx::execution::experimental::task_group tasks_`

`threads::thread_id_type id_`

`ExPolicy policy_`

`class task_canceled_exception : public exception`

`#include <task_block.hpp>` The class `task_canceled_exception` defines the type of objects thrown by `task_block::run` or `task_block::wait` if they detect that an exception is pending within the current parallel region.

Public Functions

`task_canceled_exception()`

`namespace hpx`

`namespace execution`

`namespace experimental`

`class task_group`

Public Functions

`task_group()`

`~task_group()`

`template<typename Executor, typename F, typename ...Ts>`
`void run (Executor &&exec, F &&f, Ts&&... ts)`

`template<typename F, typename ...Ts>`
`void run (F &&f, Ts&&... ts)`

`void wait ()`

`void add_exception (std::exception_ptr p)`

Private Types

```
using shared_state_type = lcos::detail::future_data<void>
```

Private Functions

```
void serialize(serialization::input_archive&, unsigned const)
void serialize(serialization::output_archive&, unsigned const)
```

Private Members

```
hpx::lcos::local::latch latch_
hpx::intrusive_ptr<shared_state_type> state_
hpx::exception_list errors_
std::atomic<bool> has_arrived_
```

Friends

```
friend hpx::execution::experimental::serialization::access
struct on_exit
```

Public Functions

```
on_exit(task_group &tg)
~on_exit()
on_exit(on_exit const &rhs)
on_exit &operator=(on_exit const &rhs)
on_exit(on_exit &&rhs)
on_exit &operator=(on_exit &&rhs)
```

Public Members

```
hpx::lcos::local::latch *latch_
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
std::enable_if<hpx::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter2>::type>
```

Assigns each value in the range given by result its corresponding element in the range [first, last] and the one preceding it except *result, which is assigned *first

The difference operations in the parallel *adjacent_difference* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly (last - first) - 1 application of the binary operator and (last - first) assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used for the input range (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- *dest*: Refers to the beginning of the sequence of elements the results will be assigned to.

The difference operations in the parallel *adjacent_difference* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *adjacent_find* is available if the user decides to provide their algorithm their own binary predicate *op*. Assigns each value in the range given by result its corresponding element in the range [first, last] and the one preceding it except *result, which is assigned *first

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.

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.

Variables

```
std::enable_if<hpx::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter2>::type>
std::enable_if<hpx::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter2>::type>
```

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Pred = detail::equal_to>
FwdIter adjacent_find(FwdIter first, FwdIter last, Pred &&pred = Pred())
```

Searches the range [first, last) for two consecutive identical elements.

Note Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Return The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

Template Parameters

- **FwdIter**: The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of an optional function/function object to use.

Parameters

- **first**: Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **pred**: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
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*.

```
template<typename ExPolicy, typename FwdIter, typename Pred = detail::equal_to>
std::enable_if<hpx::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter>::type>;
```

Searches the range [first, last) for two consecutive identical elements. This version uses the given binary predicate *pred*

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **pred**: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *adjacent_find* is available if the user decides to provide their algorithm their own binary predicate *pred*.

Return The *adjacent_find* algorithm returns a *hpx::future<InIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *InIter* otherwise. The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type none_of(ExPolicy &&policy, FwdIter first,
                                                               FwdIter last, F &&f, Proj &&proj =
                                                               Proj())
```

Checks if unary predicate *f* returns true for no elements in the range [first, last).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most *last - first* applications of the predicate *f*

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *none_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

```
template<typename ExPolicy, typename FwdIter, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type any_of(ExPolicy &&policy, FwdIter first, FwdIter
last, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for at least one element in the range [first, last).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most *last - first* applications of the predicate *f*

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *any_of* requires **F** to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The type **Type** must be such that an object of type **FwdIter** can be dereferenced and then implicitly converted to **Type**.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *any_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

```
template<typename ExPolicy, typename FwdIter, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type all_of (ExPolicy &&policy, FwdIter first, FwdIter last, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for all elements in the range [*first*, *last*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most *last - first* applications of the predicate *f*

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *all_of* requires **F** to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `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 `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.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type copy(ExPolicy      &&policy,
                                                               FwdIter1 first, FwdIter1
                                                               last, FwdIter2 dest)
```

Copies the elements in the range, defined by [first, last), to another range beginning at `dest`.

The assignments in the parallel `copy` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The assignments in the parallel `copy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `copy` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `copy` algorithm returns the pair of the input iterator `last` and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type copy_n(ExPolicy      &&policy,
                           FwdIter1   first,    Size
                           count,    FwdIter2 dest)
```

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest.

The assignments in the parallel `copy_n` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` assignments, if `count > 0`, no assignments otherwise.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Size`: The type of the argument specifying the number of elements to apply `f` to.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at `first` the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The assignments in the parallel `copy_n` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `copy_n` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `copy` algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type copy_if(ExPolicy policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest, Pred &&pred)
```

Copies the elements in the range, defined by [first, last), to another range beginning at dest. Copies only the elements for which the predicate *f* returns true. The order of the elements that are not removed is preserved.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *f*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *copy_if* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIterB, typename FwdIterE, typename T, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, std::iterator_traits<FwdIterB>::difference_type>::type count(ExPolicy
    &&policy,
    FwdIterB
    first,
    FwdIterE
    last,
    T
    const
    &value,
    Proj
    &&proj
    =
    Proj())
```

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* comparisons.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **FwdIterB:** The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIterE:** The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T:** The type of the value to search for (deduced).
- **Proj:** The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **first:** Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last:** Refers to the end of the sequence of elements the algorithm will be applied to.
- **value:** The value to search for.
- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Note The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *count* algorithm returns a *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIterB>::difference_type*). The *count* algorithm returns the number of elements satisfying the given criteria.

```
template<typename ExPolicy, typename Iter, typename Sent, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<Iter>::difference_type>::type count_if(ExPolicy
    &&pol-
    icy,
    Iter
    first,
    Sent
    last,
    F
    &&f,
    Proj
    &&proj
    =
    Proj())
```

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note Complexity: Performs exactly *last - first* applications of the predicate.

Note The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *count_if* algorithm returns *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIterB>::difference_type*). The *count* algorithm returns the number of elements satisfying the given criteria.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Iter:** The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent:** The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F:** The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj:** The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter>
util::detail::algorithm_result<ExPolicy>::type destroy (ExPolicy &&policy, FwdIter first, FwdIter last)
Destroy 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.

namespace hpx

namespace ranges

Functions

`template<typename InIter1, typename InIter2, typename Pred, typename Proj1, typename Proj2>`
`bool ends_with(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Pred &&pred, Proj1 &&proj1, Proj2 &&proj2)`

Checks whether the second range defined by [first1, last1) matches the suffix of the first range defined by [first2, last2)

The assignments in the parallel `ends_with` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear: at most `min(N1, N2)` applications of the predicate and both projections.

Template Parameters

- `InIter1`: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.

- `InIter2`: The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- `Pred`: The binary predicate that compares the projected elements.
- `Proj1`: The type of an optional projection function for the source range. This defaults to `util::projection_identity`
- `Proj1`: The type of an optional projection function for the destination range. This defaults to `util::projection_identity`

Parameters

- `first1`: Refers to the beginning of the source range.
- `last1`: Refers to the end of the source range.
- `first2`: Refers to the beginning of the destination range.
- `last2`: Refers to the end of the destination range.
- `pred`: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by `proj1` and `proj2` respectively.
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Return The `ends_with` algorithm returns `bool`. The `ends_with` algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred, typename Proj1, typename Proj2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy    &&policy,
                                                               FwdIter1 first1,
                                                               FwdIter1 last1,
                                                               FwdIter2 first2,
                                                               FwdIter2 last2,
                                                               Pred      &&pred,
                                                               Proj1    &&proj1,
                                                               Proj2    &&proj2)
```

Checks whether the second range defined by $[first1, last1]$ matches the suffix of the first range defined by $[first2, last2]$

The assignments in the parallel `ends_with` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Linear: at most $\min(N1, N2)$ applications of the predicate and both projections.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- `Pred`: The binary predicate that compares the projected elements.
- `Proj1`: The type of an optional projection function for the source range. This defaults to `util::projection_identity`
- `Proj1`: The type of an optional projection function for the destination range. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the source range.

- `last1`: Refers to the end of the source range.
- `first2`: Refers to the beginning of the destination range.
- `last2`: Refers to the end of the destination range.
- `pred`: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the two ranges projected by `proj1` and `proj2` respectively.
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel `ends_with` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `ends_with` algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `bool` otherwise. The `ends_with` algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result<ExPolicy, bool>::type equal(ExPolicy &&policy, FwdIter1 first1,
                                                       FwdIter1 last1, FwdIter2 first2, FwdIter2
                                                       last2, Pred &&op = Pred())
```

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise.

The comparison operations in the parallel `equal` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: At most $\min(\text{last1} - \text{first1}, \text{last2} - \text{first2})$ 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>
util::detail::algorithm_result<ExPolicy, bool>::type equal(ExPolicy &&policy, FwdIter1 first1,
                                                       FwdIter1 last1, FwdIter2 first2, Pred &&op
                                                       = Pred())
```

Returns true if the range `[first1, last1)` is equal to the range starting at `first2`, and false otherwise.

The comparison operations in the parallel `equal` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: At most `last1 - first1` applications of the predicate `f`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `equal` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::equal_to<>`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2**: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **op**: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

The comparison operations in the parallel `equal` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The two ranges are considered equal if, for every iterator `i` in the range `[first1, last1)`, `*i` equals `*(first2 + (i - first1))`. This overload of `equal` uses operator`==` to determine if two elements are equal.

Return The `equal` algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `bool` otherwise. The `equal` algorithm returns true if the elements in the two ranges are equal, otherwise it returns false.

```
namespace hpx
```

Functions

```
template<typename InIter, typename OutIter, typename T>
OutIter exclusive_scan(InIter first, InIter last, OutIter dest, T init)
```

Assigns through each iterator `i` in `[result, result + (last - first))` the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, `*first, ..., *(first + (i - result) - 1)`)

The reduce operations in the parallel `exclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

Template Parameters

- **InIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `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 difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i th input element in the i th sum.

Return The `exclusive_scan` algorithm returns `OutIter`. The `exclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(+, 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 FwdIter1, typename FwdIter2, typename T>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type exclusive_scan (ExPolicy      &&policy,
                                                               FwdIter1 first, FwdIter1
                                                               last, FwdIter2 dest, T
                                                               init)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))`

The reduce operations in the parallel `exclusive_scan` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `init`: The initial value for the generalized sum.

The reduce operations in the parallel `exclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *exclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.

```
template<typename InIter, typename OutIter, typename T, typename Op>
OutIter exclusive_scan(InIter first, InIter last, OutIter dest, T init, Op &&op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, ..., *(first + (i - result) - 1)).

The reduce operations in the parallel *exclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *init*: The initial value for the generalized sum.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Return The *exclusive_scan* algorithm returns *OutIter*. The *exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*1, ..., *a*N) is defined as:

- *a*1 when N is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*1, ... , *a*K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*M, ..., *a*N)) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Op>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type exclusive_scan(ExPolicy      &&policy,
                                                               FwdIter1 first, FwdIter1
                                                               last, FwdIter2 dest, T init,
                                                               Op &&op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, ..., *(first + (i - result) - 1)).

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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 `exclusive_scan` algorithm returns a `hpx::future<OutIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `OutIter` otherwise. The `exclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- $a1$ when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \dots, aN))$ where $1 < K+1 = M \leq N$.

`namespace hpx`

Functions

```
template<typename ExPolicy, typename FwdIter, typename T>
util::detail::algorithm_result<ExPolicy>::type fill(ExPolicy &&policy, FwdIter first, FwdIter last, T  
value)  
Assigns the given value to the elements in the range [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*).

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter, typename T>
util::detail::algorithm_result<ExPolicy, FwdIter>::type find(ExPolicy &&policy, FwdIter first, FwdIter
last, T const &val)
```

Returns the first element in the range [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most last - first applications of the operator==().

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to find (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **val**: the value to compare the elements to

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *find* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type find_if(ExPolicy &&policy, FwdIter first,
FwdIter last, F &&f)
```

Returns the first element in the range [first, last) for which predicate *f* returns true

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most last - first applications of the predicate.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **f**: The unary predicate which returns true for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have **const** &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *find_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if* algorithm returns the first element in the range [first, last) that satisfies the predicate *f*. If no such element exists that satisfies the predicate *f*, the algorithm returns *last*.

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type find_if_not (ExPolicy &&policy, FwdIter
first, FwdIter last, F &&f)
```

Returns the first element in the range [first, last) for which predicate *f* returns false

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most *last - first* applications of the predicate.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **f**: The unary predicate which returns false for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *find_if_not* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if_not* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate *f*, the algorithm returns *last*.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result<ExPolicy, FwdIter1>::type find_end(ExPolicy &&policy, FwdIter1
                                                               first1, FwdIter1 last1, FwdIter2
                                                               first2, FwdIter2 last2, Pred &&op
                                                               = Pred())
```

Returns the last subsequence of elements [first2, last2) found in the range [first, last) using the given predicate *f* to compare elements.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most $S^*(N-S+1)$ comparisons where $S = \text{distance}(\text{first2}, \text{last2})$ 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>
util::detail::algorithm_result<ExPolicy, FwdIter1>::type find_first_of(ExPolicy &&policy,
                                                                     FwdIter1 first, FwdIter1
                                                                     last, FwdIter2 s_first,
                                                                     FwdIter2 s_last, Pred
                                                                     &&op = Pred())
```

Searches the range $[first, last]$ for any elements in the range $[s_first, s_last]$. Uses binary predicate `p` to compare elements

The comparison operations in the parallel `find_first_of` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: at most $(S*N)$ comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `equal` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::equal_to<>`
- `Proj1`: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced `FwdIter1`.
- `Proj2`: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced `FwdIter2`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `s_first`: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- `s_last`: Refers to the end of the sequence of elements of the algorithm will be searching for.
- `op`: The binary predicate which returns `true` if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced `FwdIter1` as a projection operation before the function `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced `FwdIter2` as a projection operation before the function `op` is invoked.

The comparison operations in the parallel `find_first_of` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `find_first_of` algorithm returns a `hpx::future<FwdIter1>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter1` otherwise. The `find_first_of` algorithm returns an iterator to the first element in the range `[first, last]` that is equal to an element from the range `[s_first, s_last]`. If the length of the subsequence `[s_first, s_last]` is greater than the length of the range `[first, last]`, `last` is returned. Additionally if the size of the subsequence is empty or no

subsequence is found, *last* is also returned. This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *f*.

```
namespace hpx
```

Functions

```
template<typename InIter, typename F>
F for_each(InIter first, InIter last, F &&f)
```

Applies *f* to the result of dereferencing every iterator in the range [*first*, *last*).

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *last - first* times.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Return *f*.

Template Parameters

- *InIter*: The type of the source begin and end iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- *F*: The type of the function/function object to use (deduced). *F* must meet requirements of *MoveConstructible*.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [*first*, *last*). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have *const&*. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, void>::type for_each(ExPolicy &&policy, FwdIter first,
                                                               FwdIter last, F &&f)
```

Applies *f* to the result of dereferencing every iterator in the range [*first*, *last*).

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *last - first* times.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *FwdIter*: The type of the source begin and end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have *const&*. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *for_each* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns void otherwise.

```
template<typename InIter, typename Size, typename F>
InIter for_each_n(InIter first, Size count, F &&f)
```

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *count* times.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Return *first + count* for non-negative values of *count* and *first* for negative values.

Template Parameters

- *InIter*: The type of the source begin and end iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.

- *F*: The type of the function/function object to use (deduced). *F* must meet requirements of *MoveConstructible*.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have *const&*. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename FExPolicy, FwdIter>::type for_each_n(ExPolicy &&policy, FwdIter first,
Size count, F &&f)
```

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *count* times.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have `const&`. The type `Type` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to `Type`.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `for_each_n` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. It returns `first + count` for non-negative values of `count` and `first` for negative values.

```
namespace hpx
```

Functions

```
template<typename I, typename ...Args>
void for_loop(std::decay_t<I> first, I last, Args&&... args)
```

The `for_loop` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of `for_loop` without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Requires: `I` shall be an integral type or meet the requirements of an input iterator type. The `args` parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, `f`. `f` shall meet the requirements of `MoveConstructible`.

Template Parameters

- `I`: The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- `Args`: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `args`: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies f to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the $args$ parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by $first$. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the $args$ parameter pack excluding f , an additional argument is passed to each application of f as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of f , even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of f in the input sequence.

Complexity: Applies f exactly once for each element of the input sequence.

Remarks: If f returns a result, the result is ignored.

```
template<typename ExPolicy, typename I, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type for_loop (ExPolicy &&policy, std::decay_t<I> first, I
last, Args&&... args)
```

The `for_loop` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: I shall be an integral type or meet the requirements of an input iterator type. The $args$ parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, f . f shall meet the requirements of MoveConstructible.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **I**: The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Args**: A parameter pack, 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 `hpx::execution::sequenced_task_policy` or `hpx::execution::parallel_task_policy` and returns `void` otherwise.

```
template<typename I, typename S, typename ...Args>
void for_loop_strided(std::decay_t<I> first, I last, S stride, Args&&... args)
```

The `for_loop_strided` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of `for_loop` without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of `MoveConstructible`.

Template Parameters

- *I*: The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- *S*: The type of the stride variable. This should be an integral type.
- *Args*: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `stride`: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if `I` has integral type or meets the requirements of a bidirectional iterator.
- `args`: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by `[first, last)` should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies `f` to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the `args` parameter pack. The length of the input sequence is `last - first`.

The first element in the input sequence is specified by `first`. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the `args` parameter pack excluding `f`, an additional argument is passed to each application of `f` as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of `f`, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of `f` in the input sequence.

Complexity: Applies `f` exactly once for each element of the input sequence.

Remarks: If `f` returns a result, the result is ignored.

```
template<typename ExPolicy, typename I, typename S, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type for_loop_strided(ExPolicy           &&policy,
                                                               std::decay_t<I> first, I last, S
                                                               stride, Args&&... args)
```

The `for_loop_strided` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: `I` shall be an integral type or meet the requirements of an input iterator type. The `args` parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, `f`. `f` shall meet the requirements of `MoveConstructible`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `I`: The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- `S`: The type of the stride variable. This should be an integral type.
- `Args`: A parameter pack, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `stride`: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if `I` has integral type or meets the requirements of a bidirectional iterator.
- `args`: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last] should expose a signature equivalent to:

```
<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 `hpx::execution::sequenced_task_policy` or `hpx::execution::parallel_task_policy` and returns `void` otherwise.

```
template<typename I, typename Size, typename ...Args>
void for_loop_n(I first, Size size, Args&&... args)
```

The `for_loop_n` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of `for_loop_n` without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Template Parameters

- *I*: The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- *Size*: The type of a non-negative integral value specifying the number of items to iterate over.
- *Args*: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *size*: Refers to the number of items the algorithm will be applied to.
- *args*: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last] should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies f exactly once for each element of the input sequence.

Remarks: If f returns a result, the result is ignored.

```
template<typename ExPolicy, typename I, typename Size, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type for_loop_n(ExPolicy &&policy, I first, Size size,
                                                Args&... args)
```

The `for_loop_n` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: I shall be an integral type or meet the requirements of an input iterator type. The `args` parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, f . f shall meet the requirements of `MoveConstructible`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `I`: The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- `Size`: The type of a non-negative integral value specifying the number of items to iterate over.
- `Args`: A parameter pack, 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.
- `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 `hpx::execution::sequenced_task_policy` or `hpx::execution::parallel_task_policy` and returns `void` otherwise.

```
template<typename I, typename Size, typename S, typename ...Args>
void for_loop_n_strided(I first, Size size, S stride, Args&&... args)
```

The `for_loop_n_strided` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of `for_loop` without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Requires: I shall be an integral type or meet the requirements of an input iterator type. The $args$ parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, f . f shall meet the requirements of `MoveConstructible`.

Template Parameters

- I : The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- $Size$: The type of a non-negative integral value specifying the number of items to iterate over.
- S : The type of the stride variable. This should be an integral type.
- $Args$: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- $first$: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- $size$: Refers to the number of items the algorithm will be applied to.
- $stride$: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- $args$: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last] should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies f to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the args parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by first . Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the args parameter pack excluding f , an additional argument is passed to each application of f as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of f , even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of f in the input sequence.

Complexity: Applies f exactly once for each element of the input sequence.

Remarks: If f returns a result, the result is ignored.

```
template<typename ExPolicy, typename I, typename Size, typename S, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type for_loop_n_strided(ExPolicy &&policy, I first, Size
size, S stride, Args&&... args)
```

The `for_loop_n_strided` implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: I shall be an integral type or meet the requirements of an input iterator type. The args parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, f . f shall meet the requirements of `MoveConstructible`.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **I**: The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Size**: The type of a non-negative integral value specifying the number of items to iterate over.
- **S**: The type of the stride variable. This should be an integral type.
- **Args**: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **size**: Refers to the number of items the algorithm will be applied to.
- **stride**: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.

- *args*: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Return The `for_loop_n_strided` algorithm returns a `hpx::future<void>` if the execution policy is of type `hpx::execution::sequenced_task_policy` or `hpx::execution::parallel_task_policy` and returns `void` otherwise.

```
namespace hpx
```

```
namespace parallel
```

Functions

```
template<typename T>
constexpr detail::induction_stride_helper<T> induction(T &&value, std::size_t stride)
```

The function template returns an induction object of unspecified type having a value type and encapsulating an initial value *value* of that type and, optionally, a stride.

For each element in the input range, a looping algorithm over input sequence *S* computes an induction value from an induction variable and ordinal position *p* within *S* by the formula *i* + *p* * *stride* if a stride was specified or *i* + *p* otherwise. This induction value is passed to the element access function.

If the *value* argument to *induction* is a non-const lvalue, then that lvalue becomes the live-out object for the returned induction object. For each induction object that has a live-out object, the looping

algorithm assigns the value of $i + n * \text{stride}$ 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)

```
namespace hpx
```

```
namespace parallel
```

Functions

```
template<typename T, typename Op>
constexpr detail::reduction_helper<T, typename std::decay<Op>::type> reduction(T
    &var,
    T
    const
    &iden-
    tity,
    Op
    &&com-
    biner)
```

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 hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type generate (ExPolicy &&policy, FwdIter first,
                                                               FwdIter last, F &&f)
```

Assign each element in range [first, last) a value generated by the given function object f

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter:** The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F:** The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **first:** Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last:** Refers to the end of the sequence of elements the algorithm will be applied to.
- **f:** generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type generate_n(ExPolicy &&policy, FwdIter first,
Size count, F &&f)
```

Assigns each element in range [first, first+count) a value generated by the given function object g.

The assignments in the parallel *generate_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *count* invocations of *f* and assignments, for *count* > 0.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter:** The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F:** The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **first:** Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count:** Refers to the number of elements in the sequence the algorithm will be applied to.
- **f:** Refers to the generator function object that will be called. The signature of the function should be equivalent to

```
Ret fun();
```

The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

The assignments in the parallel *generate_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::less>
util::detail::algorithm_result<ExPolicy, bool>::type::includes(ExPolicy &&policy, FwdIter1
first1, FwdIter1 last1, FwdIter2
first2, FwdIter2 last2, Pred &&op
= Pred())
```

Returns true if every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note At most $2*(N1+N2-1)$ comparisons, where $N1 = \text{std::distance(first1, last1)}$ and $N2 = \text{std::distance(first2, last2)}$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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.

```
namespace hpx
```

Functions

```
template<typename InIter, typename OutIter>
OutIter inclusive_scan(InIter first, InIter last, OutIter dest)
Assigns through each iterator i in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, *first, ..., *(first + (i - result))).
```

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *inclusive_scan* algorithm returns *OutIter*. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a₁, ..., a_N) is defined as:

- a₁ when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a₁, ..., a_K)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, a_M, ..., a_N) where 1 < K+1 = M <= N.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
```

```
util::detail::algorithm_result<ExPolicy, FwdIter2>::type inclusive_scan(ExPolicy      &&policy,
                                                               FwdIter1 first, FwdIter1
                                                               last, FwdIter2 dest)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM($+, *first, \dots, *(first + (i - result))$).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the i th input element in the i th sum.

Return The *inclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM($+, a_1, \dots, a_N$) is defined as:

- a_1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM($+, a_1, \dots, a_K$)
 - GENERALIZED_NONCOMMUTATIVE_SUM($+, a_M, \dots, a_N$) where $1 < K+1 = M \leq N$.

```
template<typename InIter, typename OutIter, typename Op>
OutIter inclusive_scan(InIter first, InIter last, OutIter dest, Op &&op)

Assigns through each iterator  $i$  in  $[result, result + (last - first))$  the value of GENERALIZED_NONCOMMUTATIVE_SUM( $op, *first, \dots, *(first + (i - result))$ ).
```

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *inclusive_scan* algorithm returns *OutIter*. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename OpExPolicy, FwdIter2>::type inclusive_scan(ExPolicy &&policy,
                                                               FwdIter1 first, FwdIter1
                                                               last, FwdIter2 dest, Op
                                                               &&op)
```

Assigns through each iterator *i* in [*result*, *result* + (*last* - *first*)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(*op*, **first*, ..., *(*first* + (*i* - *result*))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Op`: The type of the binary function object used for the reduction operation.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel `inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i th input element in the i th sum.

Return The `inclusive_scan` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `inclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN)` is defined as:

- a_1 when N is 1
- $\text{GENERALIZED_NONCOMMUTATIVE_SUM}(\text{op}, a_1, \dots, a_K)$
 - $\text{GENERALIZED_NONCOMMUTATIVE_SUM}(+, a_M, \dots, a_N)$ where $1 < K+1 = M \leq N$.

```
template<typename InIter, typename OutIter, typename T, typename Op>
OutIter inclusive_scan(InIter first, InIter last, OutIter dest, Op &&op, T init)
Assigns through each iterator  $i$  in  $[result, result + (last - first))$  the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, ..., *(first + (i - result))).
```

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Op*: The type of the binary function object used for the reduction operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- *init*: The initial value for the generalized sum.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Return The *inclusive_scan* algorithm returns *OutIter*. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., 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, typename Op>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type inclusive_scan(ExPolicy &&policy,
                                                               FwdIter1 first, FwdIter1
                                                               last, FwdIter2 dest, T init,
                                                               Op &&op)
```

Assigns through each iterator *i* in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, ..., *(first + (i - result)))`.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).
- `Op`: The type of the binary function object used for the reduction operation.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `init`: The initial value for the generalized sum.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel `inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i th input element in the i th sum. If `op` is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

Return The `inclusive_scan` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `inclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- $a1$ when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \dots, aN))$ where $1 < K+1 = M \leq N$.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename RandIter, typename Comp = detail::less>
util::detail::algorithm_result<ExPolicy, bool>::type is_heap (ExPolicy &&policy, RandIter first, Ran-
dIter last, Comp &&comp = Comp())
```

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp**: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *is_heap* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

```
template<typename ExPolicy, typename RandIter, typename Comp = detail::less>
util::detail::algorithm_result<ExPolicy, RandIter>::type is_heap_until (ExPolicy &&policy, Ran-
dIter first, RandIter last,
Comp &&comp = Comp())
```

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [*first*, *it*] is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `RandIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- `Comp`: The type of the function/function object to use (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `comp`: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

The application of function objects in parallel algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `is_heap_until` algorithm returns a `hpx::future<RandIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `RandIter` otherwise. The `is_heap_until` algorithm returns the upper bound of the largest range beginning at `first` which is a max heap. That is, the last iterator `it` for which range `[first, it]` is a max heap.

namespace hpx

Functions

```
template<typename FwdIter, typename Pred>
bool is_partitioned(FwdIter first, FwdIter last, Pred &&pred)
```

Determines if the range `[first, last)` is partitioned.

Note Complexity: at most (N) predicate evaluations where $N = \text{distance}(\text{first}, \text{last})$.

Return The `is_partitioned` algorithm returns `bool`. The `is_partitioned` algorithm returns true if each element in the sequence for which `pred` returns true precedes those for which `pred` returns false. Otherwise `is_partitioned` returns false. If the range `[first, last)` contains less than two elements, the function is always true.

Template Parameters

- `FwdIter`: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- `Pred`: The type of the function/function object to use (deduced).

Parameters

- *first*: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- *pred*: Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

```
template<typename ExPolicy, typename FwdIter, typename Pred>
util::detail::algorithm_result<ExPolicy, bool>::type is_partitioned(ExPolicy &&policy, FwdIter
first, FwdIter last, Pred
&&pred)
```

Determines if the range [first, last) is partitioned.

The predicate operations in the parallel *is_partitioned* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: at most (*N*) predicate evaluations where *N* = $\text{distance}(\text{first}, \text{last})$.

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.
- *Pred*: The type of the function/function object to use (deduced). *Pred* must be *CopyConstructible* when using a parallel policy.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- *pred*: Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

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.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Pred = hpx::parallel::v1::detail::less>
bool is_sorted(FwdIter first, FwdIter last, Pred &&pred = Pred())
```

Determines if the range [first, last) is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Pred*: The type of an optional function/function object to use.

Parameters

- *first*: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- *pred*: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

<code>bool pred(const Type &a, const Type &b);</code>

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Return The *is_sorted* algorithm returns a *bool*. The *is_sorted* algorithm returns true if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

```
template<typename ExPolicy, typename FwdIter, typename Pred = hpx::parallel::v1::detail::less>
hpx::parallel::util::algorithm_result<ExPolicy, bool>::type is_sorted(ExPolicy      &&policy,
                                                               FwdIter first, FwdIter
                                                               last, Pred &&pred =
                                                               Pred())
```

Determines if the range [first, last) is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`

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 $[\text{first}, \text{last}]$ satisfies the predicate passed. If the range $[\text{first}, \text{last}]$ contains less than two elements, the function always returns true.

```
template<typename FwdIter, typename Pred = hpx::parallel::v1::detail::less>
FwdIter is_sorted_until (FwdIter first, FwdIter last, Pred &&pred = Pred())
    Returns the first element in the range  $[\text{first}, \text{last}]$  that is not sorted. Uses a predicate to compare elements or the less than operator.
```

The comparison operations in the parallel *is_sorted_until* algorithm execute in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- **FwdIter**: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred**: The type of an optional function/function object to use.

Parameters

- *first*: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- *pred*: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Return The *is_sorted_until* algorithm returns a *FwdIter*. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, *last* is returned.

```
template<typename ExPolicy, typename FwdIter, typename Pred = hpx::parallel::v1::detail::less>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type is_sorted_until(ExPolicy
    &&policy,
    FwdIter
    first,
    FwdIter
    last, Pred
    &&pred =
    Pred())
```

Returns the first element in the range [*first*, *last*) that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted_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.

```
namespace hpx
```

Functions

```
template<typename InIter1, typename InIter2, typename Pred>
bool lexicographical_compare(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Pred
    &&pred)
```

Checks if the first range $[first1, last1]$ is lexicographically less than the second range $[first2, last2]$. uses a provided predicate to compare elements.

The comparison operations in the parallel `lexicographical_compare` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std}::\text{distance}(first1, last1)$ and $N2 = \text{std}::\text{distance}(first2, last2)$.

Template Parameters

- `InIter1`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input iterator.
- `InIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an input iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `lexicographical_compare` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`

Parameters

- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `pred`: Refers to the comparison function that the first and second ranges will be applied to

Note Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Return The *lexicographically_compare* algorithm returns a returns *bool* if the execution policy object is not passed in. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2], it returns false.

```
template<typename FwdIter1, typename FwdIter2, typename Pred>
util::detail::algorithm_result<ExPolicy, bool>::type lexicographical_compare (ExPolicy &&policy,
                                                               FwdIter1 first1, FwdIter1 last1,
                                                               FwdIter2 first2, FwdIter2 last2,
                                                               Pred &&pred)
```

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std}::\text{distance}(\text{first1}, \text{last1})$ and $N2 = \text{std}::\text{distance}(\text{first2}, \text{last2})$.

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 *CopyConstructible*. This defaults to *std::less<>*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.

- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `pred`: Refers to the comparison function that the first and second ranges will be applied to

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Return The *lexicographically_compare* algorithm returns a `hpx::future<bool>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `bool` otherwise. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range `[first2, last2)`, it returns false.

namespace hpx

Functions

```
template<typename ExPolicy, typename RndIter, typename Comp>
util::detail::algorithm_result<ExPolicy>::type make_heap(ExPolicy &&policy, RndIter first, RndIter
last, Comp &&comp)
```

Constructs a *max heap* in the range `[first, last)`.

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

Note Complexity: at most $(3*N)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$.

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.
- `RndIter`: The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- `comp`: Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

```
bool comp(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type` must be such that objects of types `RndIter` can be dereferenced and then implicitly converted to `Type`.

The comparison operations in the parallel `make_heap` algorithm invoked with an execution policy object of type `parallel_execution_policy` or `parallel_task_execution_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `make_heap` algorithm returns a `hpx::future<void>` if the execution policy is of type `task_execution_policy` and returns `void` otherwise.

```
template<typename ExPolicy, typename RndIter>
hpx::parallel::util::detail::algorithm_result<ExPolicy>::type make_heap(ExPolicy &&policy, RndIter
first, RndIter last)
Constructs a max heap in the range [first, last). Uses the operator < for comparisons.
```

The predicate operations in the parallel `make_heap` algorithm invoked with an execution policy object of type `sequential_execution_policy` executes in sequential order in the calling thread.

Note Complexity: at most $(3*N)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$.

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.
- `RndIter`: The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of that the algorithm will be applied to.

The comparison operations in the parallel `make_heap` algorithm invoked with an execution policy object of type `parallel_execution_policy` or `parallel_task_execution_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `make_heap` algorithm returns a `hpx::future<void>` if the execution policy is of type `task_execution_policy` and returns `void` otherwise.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename RandIter1, typename RandIter2, typename RandIter3, typename Comp = std::less<>,
         util::detail::algorithm_result<ExPolicy, RandIter3>::type> merge(ExPolicy &&policy, RandIter1 first1,
                                                                           RandIter1 last1, RandIter2 first2, RandIter2 last2, RandIter3 dest, Comp
                                                                           &&comp = Comp())
```

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at dest. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs $O(\text{std}::\text{distance}(\text{first1}, \text{last1}) + \text{std}::\text{distance}(\text{first2}, \text{last2}))$ 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<>`

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*

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *merge* algorithm returns a *hpx::future<RandIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandIter3* otherwise. The *merge* algorithm returns the destination iterator to the end of the *dest* range.

```
template<typename ExPolicy, typename RandIter, typename Comp = std::less>
util::detail::algorithm_result<ExPolicy>::type inplace_merge(ExPolicy &&policy, RandIter first,
                                                               RandIter middle, RandIter last, Comp
                                                               &&comp = Comp())
```

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs $O(\text{std}::\text{distance}(\text{first}, \text{last}))$ applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to *std::less*<

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the first sorted range the algorithm will be applied to.
- **middle**: Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **last**: Refers to the end of the second sorted range the algorithm will be applied to.
- **comp**: *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *inplace_merge* algorithm returns a `hpx::future<void>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns void otherwise. The *inplace_merge* algorithm returns the source iterator *last*

namespace hpx

Functions

```
template<typename FwdIter, typename F>
FwdIter min_element(FwdIter first, FwdIter last, F &&f)
```

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std}:\text{distance}(first, last)$.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `F`: The type of the function/function object to use (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the 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`.

Return The *min_element* algorithm returns `FwdIter`. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns `last` if the range is empty.

```
template<typename ExPolicy, typename FwdIter, typename T>
util::detail::algorithm_result<ExPolicy, FwdIter>::type min_element(ExPolicy &&policy, FwdIter
first, FwdIter last, F &&f)
```

Finds the smallest element in the range [first, last) using the given comparison function *f*.

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 $N = \text{std}:\text{distance}(first, last)$.

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 (deduced). This iterator type must meet the requirements of a forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `min_element` requires `F` to meet the requirements of `CopyConstructible`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the 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`.

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 FwdIter, typename F>
FwdIter max_element (FwdIter first, FwdIter last, F &&f)
```

Finds the largest element in the range `[first, last)` using the given comparison function `f`.

The comparisons in the parallel `min_element` algorithm execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std}:\text{distance}(first, last)$.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `F`: The type of the function/function object to use (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the 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`.

Return The `max_element` algorithm returns `FwdIter`. The `max_element` algorithm returns the iterator to the smallest element in the range `[first, last)`. If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns `last` if the range is empty.

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type max_element(ExPolicy &&policy, FwdIter
first, FwdIter last, F &&f)
    Removes all elements satisfying specific criteria from the range Finds the largest element in the range
    [first, last) using the given comparison function f.
```

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 $N = \text{std}:\text{distance}(first, last)$.

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 (deduced). This iterator type must meet the requirements of a forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `max_element` requires `F` to meet the requirements of `CopyConstructible`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the This argument is optional and defaults to `std::less`. The left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

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 FwdIter, typename F>
minmax_element_result<FwdIter> minmax_element (FwdIter first, FwdIter last, F &&f)
    Finds the largest element in the range [first, last) using the given comparison function f.
```

The comparisons in the parallel *minmax_element* algorithm execute in sequential order in the calling thread.

Note Complexity: At most $\max(\text{floor}(3/2*(N-1)), 0)$ applications of the predicate, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- *F*: The type of the function/function object to use (deduced).

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *f*: The binary predicate which returns true if the 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*.

Return The *minmax_element* algorithm returns a *minmax_element_result<FwdIter>*. The *minmax_element* algorithm returns a pair consisting of an iterator to the smallest element as the min element and an iterator to the largest element as the max element. Returns *minmax_element_result<FwdIter>{first, first}* if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, minmax_element_result<FwdIter>>::type minmax_element (ExPolicy
    &&policy,
    FwdIter
    first,
    FwdIter
    last,
    F
    &&f)
```

Finds the largest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $\max(\text{floor}(3/2*(N-1)), 0)$ applications of the predicate, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$.

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 (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 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 *minmax_element_result<FwdIter>* *<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *minmax_element_result<FwdIter>* otherwise. The *minmax_element* algorithm returns a pair consisting of an iterator to the smallest element as the min element and an iterator to the largest element as the max element. Returns *std::make_pair(first, first)* if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

namespace hpx

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch (ExPolicy &&policy,
                                                               FwdIter1
                                                               first1,   FwdIter1
                                                               last1,   FwdIter2
                                                               first2,   FwdIter2
                                                               last2,    Pred
                                                               &&op = Pred())
```

Returns true if the range [**first1**, **last1**) is mismatch to the range [**first2**, **last2**), and false otherwise.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $\min(\text{last1} - \text{first1}, \text{last2} - \text{first2})$ applications of the predicate *f*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and $(\text{last1} - \text{first1}) \neq (\text{last2} - \text{first2})$ then no applications of the predicate *f* are made.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2**: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2**: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op**: The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The two ranges are considered mismatch if, for every iterator *i* in the range [first1, last1), **i* mismatchs **(first2 + (i - first1))*. This overload of mismatch uses operator== to determine if two elements are mismatch.

Return The *mismatch* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *mismatch* algorithm returns true if the elements in the two ranges are mismatch, otherwise it returns false. If the length of the range [first1, last1) does not mismatch the length of the range [first2, last2), it returns false.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch(ExPolicy &&policy,
                                                               FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, Pred pred
                                                               &&op = Pred())
```

Returns std::pair with iterators to the first two non-equivalent elements.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most *last1 - first1* applications of the predicate *f*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to<>*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last1*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *first2*: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- *op*: The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *mismatch* algorithm returns a `hpx::future<std::pair<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `std::pair<FwdIter1, FwdIter2>` otherwise. The *mismatch* algorithm returns the first mismatching pair of elements from two ranges: one defined by `[first1, last1)` and another defined by `[first2, last2)`.

namespace hpx

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type move(ExPolicy &&policy, FwdIter1 first,
                                                               FwdIter1 last, FwdIter2 dest)
Moves the elements in the range [first, last), to another range beginning at dest. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.
```

The move assignments in the parallel *move* algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` move assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the move assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The move assignments in the parallel *move* algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *move* algorithm returns a `hpx::future<FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The *move* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

```
namespace hpx
```

Functions

```
template<typename RandomIt, typename Pred>
void nth_element(RandomIt first, RandomIt nth, RandomIt last, Pred &&pred)
```

`nth_element` is a partial sorting algorithm that rearranges elements in $[first, last]$ such that the element pointed at by `nth` is changed to whatever element would occur in that position if $[first, last]$ were sorted and all of the elements before this new `nth` element are less than or equal to the elements after the new `nth` element.

The comparison operations in the parallel `nth_element` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear in `std::distance(first, last)` on average. $O(N)$ applications of the predicate, and $O(N \log N)$ swaps, where $N = last - first$.

Template Parameters

- `RandomIt`: The type of the source begin, `nth`, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- `Pred`: Comparison function object which returns true if the first argument is less than the second. This defaults to `std::less<>`.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `nth`: Refers to the iterator defining the sort partition point
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `pred`: Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type must be such that an object of type `randomIt` can be dereferenced and then implicitly converted to `Type`. This defaults to `std::less<>`.

Return The `nth_element` algorithms returns nothing.

```
template<typename ExPolicy, typename RandomIt, typename Pred>
void nth_element(ExPolicy &&policy, RandomIt first, RandomIt nth, RandomIt last, Pred &&pred)
```

`nth_element` is a partial sorting algorithm that rearranges elements in $[first, last]$ such that the element pointed at by `nth` is changed to whatever element would occur in that position if $[first, last]$ were sorted and all of the elements before this new `nth` element are less than or equal to the elements after the new `nth` element.

The comparison operations in the parallel *nth_element* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *RandomIt*: The type of the source begin, nth, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- *Pred*: Comparison function object which returns true if the first argument is less than the second. This defaults to std::less<>.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *nth*: Refers to the iterator defining the sort partition point
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.

The assignments in the parallel *nth_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *nth_element* algorithms returns nothing.

```
namespace hpx
```

Functions

```
template<typename RandIter, typename Comp>
void partial_sort(RandIter first, RandIter middle, RandIter last, Comp &&comp = Comp())
Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.
```

Note Complexity: Approximately (last - first) * log(middle - first) comparisons.

Return The *partial_sort* algorithm returns nothing.

Template Parameters

- **RandIter**: The type of the source begin, middle, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced). Comp defaults to `detail::less`.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle**: Refers to the middle of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp**: comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. It defaults to `detail::less`.

```
template<typename ExPolicy, typename RandIter, typename Comp>
util::detail::algorithm_result_t<ExPolicy> partial_sort (ExPolicy &&policy, RandIter first, RandIter
                                                               middle, RandIter last, Comp &&comp =
                                                               Comp() )
```

Places the first $\text{middle} - \text{first}$ elements from the range $[\text{first}, \text{last}]$ as sorted with respect to comp into the range $[\text{first}, \text{middle}]$. The rest of the elements in the range $[\text{middle}, \text{last}]$ are placed in an unspecified order.

Note Complexity: Approximately $(\text{last} - \text{first}) * \log(\text{middle} - \text{first})$ comparisons.

Return The `partial_sort` algorithm returns a `hpx::future<void>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns void otherwise.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **RandIter**: The type of the source begin, middle, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced). Comp defaults to `detail::less`.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle**: Refers to the middle of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp**: comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. It defaults to `detail::less`.

```
namespace hpx
```

Functions

```
template<typename InIter, typename RandIter, typename Comp>
RandIter partial_sort_copy(InIter first, InIter last, RandIter d_first, RandIter d_last, Comp
                           &&comp = Comp())
Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range
[d_first, d_last). At most d_last - d_first of the elements are placed sorted to the range [d_first, d_first + n)
where n is the number of elements to sort (n = min(last - first, d_last - d_first)).
```

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ and $D = \text{std}::\text{distance}(\text{d_first}, \text{d_last})$ comparisons.

Template Parameters

- **InIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **RandIter**: The type of the destination iterators used(deduced) This iterator type must meet the requirements of a random iterator.
- **Comp**: The type of the function/function object to use (deduced). Comp defaults to `detail::less`.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **d_first**: Refers to the beginning of the destination range.
- **d_last**: Refers to the end of the destination range.
- **comp**: comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to `detail::less`.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partial_sort_copy* algorithm returns a returns *RandomIt*. The algorithm returns an iterator to the element defining the upper boundary of the sorted range i.e. $d_{\text{first}} + \min(\text{last} - \text{first}, \text{d}_{\text{last}} - \text{d}_{\text{first}})$

```
template<typename ExPolicy, typename FwdIter, typename RandIter, typename Comp>
parallel::util::detail::algorithm_result_t<ExPolicy, RandIter> partial_sort_copy(ExPolicy
                           &&policy,
                           FwdIter first,
                           FwdIter last,
                           RandIter d_first,
                           RandIter d_last,
                           Comp &&comp
                           = Comp())
Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range
```

[d_first, d_last). At most d_last - d_first of the elements are placed sorted to the range [d_first, d_first + n) where n is the number of elements to sort (n = min(last - first, d_last - d_first)).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ and $D = \text{std}::\text{distance}(d_{\text{first}}, d_{\text{last}})$ comparisons.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *RandIter*: The type of the destination iterators used(deduced) This iterator type must meet the requirements of an random iterator.
- *Comp*: The type of the function/function object to use (deduced). *Comp* defaults to *detail::less*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *d_first*: Refers to the beginning of the destination range.
- *d_last*: Refers to the end of the destination range.
- *comp*: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator. This defaults to *detail::less*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partial_sort_copy* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator to the element defining the upper boundary of the sorted range i.e. $d_{\text{first}} + \min(\text{last} - \text{first}, d_{\text{last}} - d_{\text{first}})$

namespace hpx

Functions

template<typename **FwdIter**, typename **Pred**, typename **Proj**>
FwdIter partition (ExPolicy *&&policy*, FwdIter *first*, FwdIter *last*, Pred *&&pred*, Proj *&&proj*)
 Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: At most $2 * (\text{last} - \text{first})$ swaps. Exactly *last - first* applications of the predicate and projection.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *partition* algorithm returns returns *FwdIter*. The *partition* algorithm returns the iterator to the first element of the second group.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred**, typename **Proj**>
util::detail::algorithm_result_t<ExPolicy, FwdIter> partition (ExPolicy *&&policy*, FwdIter *first*,
FwdIter last, Pred *&&pred*, Proj
&&proj)

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $2 * (\text{last} - \text{first})$ swaps. Exactly $\text{last} - \text{first}$ applications of the predicate and projection.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the 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 BidirIter, typename F, typename Proj>
BidirIter stable_partition(BidirIter first, BidirIter last, F &&f, Proj &&proj)
```

Permutes the elements in the range `[first, last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first, i)` `INVOKE(f, INVOKE(proj, *j)) != false`, and for every iterator `k` in the range `[i, last)`, `INVOKE(f, INVOKE(proj, *k)) == false`

The invocations of `f` in the parallel `stable_partition` algorithm invoked without an execution policy object executes in sequential order in the calling thread.

Note Complexity: At most $(\text{last} - \text{first}) * \log(\text{last} - \text{first})$ swaps, but only linear number of swaps if there is enough extra memory. Exactly $\text{last} - \text{first}$ applications of the predicate and projection.

Template Parameters

- **BidirIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have **const&**. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Return The *stable_partition* algorithm returns an iterator *i* such that for every iterator *j* in the range [first, *i*), *f(*j)* != false *INVOKE(f, INVOKE(proj, *j))* != false, and for every iterator *k* in the range [*i*, last), *f(*k)* == false *INVOKE(f, INVOKE (proj, *k))* == false. The relative order of the elements in both groups is preserved.

```
template<typename ExPolicy, typename BidirIter, typename F, typename Proj>
util::detail::algorithm_result_t<ExPolicy, BidirIter> stable_partition (ExPolicy &&policy,
                                                               BidirIter first, BidirIter
                                                               last, F &&f, Proj &&proj)
```

Permutes the elements in the range [first, last) such that there exists an iterator *i* such that for every iterator *j* in the range [first, *i*] *INVOKE(f, INVOKE (proj, *j))* != false, and for every iterator *k* in the range [*i*, last), *INVOKE(f, INVOKE (proj, *k))* == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory. Exactly *last - first* applications of the predicate and projection.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **BidirIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *stable_partition* algorithm returns an iterator *i* such that for every iterator *j* in the range [first, *i*), *f(*j)* != false *(INVOKE(f, INVOKE(proj, *j))* != false, and for every iterator *k* in the range [*i*, last), *f(*k)* == false *(INVOKE(f, INVOKE(proj, *k))* == false. The relative order of the elements in both groups is preserved. If the execution policy is of type *parallel_task_policy* the algorithm returns a future<> referring to this iterator.

```
template<typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
std::pair<FwdIter2, FwdIter3> partition_copy(FwdIter1 first, FwdIter1 last, FwdIter2 dest_true,
                                             FwdIter3 dest_false, Pred &&pred, Proj &&proj)
Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate pred. The elements, that satisfy the predicate pred are copied to the range beginning at dest_true. The rest of the elements are copied to the range beginning at dest_false. The order of the elements is preserved.
```

The assignments in the parallel *partition_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *f*.

Template Parameters

- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter3**: The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest_true`: Refers to the beginning of the destination range for the elements that satisfy the predicate `pred`
- `dest_false`: Refers to the beginning of the destination range for the elements that don't satisfy the predicate `pred`.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `is` invoked.

Return The `partition_copy` algorithm returns `std::pair<OutIter1, OutIter2>`. The `partition_copy` algorithm returns the pair of the destination iterator to the end of the `dest_true` range, and the destination iterator to the end of the `dest_false` range.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred, typename util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter2, FwdIter3>> partition_copy(ExPolicy
    &&policy,
    FwdIter1
    first,
    FwdIter1
    last,
    FwdIter2
    dest_true,
    FwdIter3
    dest_false,
    Pred
    &&pred,
    Proj
    &&proj)
```

Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate `pred`. The elements, that satisfy the predicate `pred`, are copied to the range beginning at `dest_true`. The rest of the elements are copied to the range beginning at `dest_false`. The order of the elements is preserved.

The assignments in the parallel `partition_copy` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs not more than `last - first` assignments, exactly `last - first` applications of the predicate `f`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range for the elements that satisfy the predicate `pred` (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter3`: The type of the iterator representing the destination range for the elements that don't satisfy the predicate `pred` (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `partition_copy` requires `Pred` to meet the requirements of `CopyConstructible`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest_true`: Refers to the beginning of the destination range for the elements that satisfy the predicate `pred`
- `dest_false`: Refers to the beginning of the destination range for the elements that don't satisfy the predicate `pred`.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `is` invoked.

The assignments in the parallel `partition_copy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `partition_copy` algorithm returns a `hpx::future<std::pair<OutIter1, OutIter2>>` if the execution policy is of type `parallel_task_policy` and returns `std::pair<OutIter1, OutIter2>` otherwise. The `partition_copy` algorithm returns the pair of the destination iterator to the end of the `dest_true` range, and the destination iterator to the end of the `dest_false` range.

namespace hpx

Functions

```
template<typename ExPolicy, typename FwdIter, typename T, typename F>
util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, FwdIter first, FwdIter
last, T init, F &&f)
>Returns GENERALIZED_SUM(f, init, *first, ..., *(first + (last - first) - 1)).
```

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: O(*last* - *first*) applications of the predicate *f*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have **const&**. The types *Type1 Ret* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to any of those types.

- **init**: The initial value for the generalized sum.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

Note GENERALIZED_SUM(*op*, *a*1, ..., *a*N) is defined as follows:

- *a*1 when N is 1
- *op*(GENERALIZED_SUM(*op*, *b*1, ..., *b*K), GENERALIZED_SUM(*op*, *b*M, ..., *b*N)), where:

- b_1, \dots, b_N may be any permutation of a_1, \dots, a_N and
- $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter, typename T>
util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, FwdIter first, FwdIter
last, T init)
>Returns GENERALIZED_SUM(+, init, *first, ..., *(first + (last - first) - 1)).
```

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the operator $+$ ().

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **init**: The initial value for the generalized sum.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum (applying operator $+$ ()) over the elements given by the input range [**first**, **last**).

Note GENERALIZED_SUM(+, a_1, \dots, a_N) is defined as follows:

- a_1 when N is 1
- $\text{op}(\text{GENERALIZED_SUM}(+, b_1, \dots, b_K), \text{GENERALIZED_SUM}(+, b_M, \dots, b_N))$, where:
 - b_1, \dots, b_N may be any permutation of a_1, \dots, a_N and
 - $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter>
```

```
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIter>::value_type>::type reduce (ExPolicy
&&policy,
FwdIter
first,
FwdIter
last)
```

Returns GENERALIZED_SUM(+, T(), *first, ..., *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the operator+().

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise (where *T* is the *value_type* of *FwdIter*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

Note The type of the initial value (and the result type) *T* is determined from the *value_type* of the used *FwdIter*.

Note GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - $1 < K+1 = M \leq N$.

```
namespace hpx
```

```
namespace parallel
```

Functions

```
template<typename ExPolicy, typename RanIter, typename RanIter2, typename FwdIter1, typename FwdIter2>
util::detail::algorithm_result<ExPolicy, util::in_out_result<FwdIter1, FwdIter2>>::type reduce_by_key (ExPolicy
    &&policy,
    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: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **RanIter:** The type of the key iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **RanIter2:** The type of the value iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **FwdIter1:** The type of the iterator representing the destination key range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2:** The type of the iterator representing the destination value range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Compare:** The type of the optional function/function object to use to compare keys (deduced). Assumed to be `std::equal_to` otherwise.
- **Func:** The type of the function/function object to use (deduced). Unlike its sequential form,

the parallel overload of `copy_if` requires F to meet the requirements of *CopyConstructible*.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `key_first`: Refers to the beginning of the sequence of key elements the algorithm will be applied to.
- `key_last`: Refers to the end of the sequence of key elements the algorithm will be applied to.
- `values_first`: Refers to the beginning of the sequence of value elements the algorithm will be applied to.
- `keys_output`: Refers to the start output location for the keys produced by the algorithm.
- `values_output`: Refers to the start output location for the values produced by the algorithm.
- `comp`: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.
- `func`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun (const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`. The types `Type1 Ret` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to any of those types.

The application of function objects in parallel algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `reduce_by_key` algorithm returns a `hpx::future<pair<Iter1,Iter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `pair<Iter1,Iter2>` otherwise.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename T>
FwdIter remove(FwdIter first, FwdIter last, T const &value)
```

Removes all elements satisfying specific criteria from the range [first, last) and returns a past-the-end iterator for the new end of the range. This version removes all elements that are equal to `value`.

The assignments in the parallel `remove` algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than $last - first$ assignments, exactly $last - first$ applications of the operator `==()`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to remove (deduced). This value type must meet the requirements of `CopyConstructible`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `value`: Specifies the value of elements to remove.

Return The `remove` algorithm returns a `FwdIter`. The `remove` algorithm returns the iterator to the new end of the range.

```
template<typename ExPolicy, typename FwdIter, typename T>
util::detail::algorithm_result<ExPolicy, FwdIter>::type remove (ExPolicy &&policy, FwdIter first,
                                                               FwdIter last, T const &value)
```

Removes all elements satisfying specific criteria from the range [`first`, `last`] and returns a past-the-end iterator for the new end of the range. This version removes all elements that are equal to `value`.

The assignments in the parallel `remove` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs not more than `last - first` assignments, exactly `last - first` applications of the operator`==()`.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to remove (deduced). This value type must meet the requirements of `CopyConstructible`.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `value`: Specifies the value of elements to remove.

The assignments in the parallel `remove` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `remove` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. The `remove` algorithm returns the iterator to the new end of the range.

```
template<typename FwdIter, typename Pred>
FwdIter remove_if (FwdIter first, FwdIter last, Pred &&pred)
```

Removes all elements satisfying specific criteria from the range [`first`, `last`] and returns a past-the-end

iterator for the new end of the range. This version removes all elements for which predicate *pred* returns true.

The assignments in the parallel *remove_if* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *pred*.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Return The *remove_if* algorithm returns a *FwdIter*. The *remove_if* algorithm returns the iterator to the new end of the range.

```
template<typename ExPolicy, typename FwdIter, typename Pred>
util::detail::algorithm_result<ExPolicy, FwdIter>::type remove_if(ExPolicy &&policy, FwdIter first,
                                                               FwdIter last, Pred &&pred)
```

Removes all elements satisfying specific criteria from the range [first, last) and returns a past-the-end iterator for the new end of the range. This version removes all elements for which predicate *pred* returns true.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *pred*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *remove_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove_if* algorithm returns the iterator to the new end of the range.

namespace hpx

Functions

```
template<typename InIter, typename OutIter, typename T>
FwdIter remove_copy (InIter first, InIter last, OutIter dest, T const &value)
```

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the comparison operator returns false when compare to *value*. The order of the elements that are not removed is preserved.

Effects: Copies all the elements referred to by the iterator *it* in the range [first, last) for which the following corresponding conditions do not hold: **it == value*

The assignments in the parallel *remove_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *f*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *T*: The type that the result of dereferencing *FwdIter1* is compared to.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `val`: Value to be removed.

Return The `remove_copy` algorithm returns an `OutIter`. The `remove_copy` algorithm returns the iterator to the element past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T>
FwdIter remove_copy (ExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest, T const
&val)
```

Copies the elements in the range, defined by [first, last), to another range beginning at dest. Copies only the elements for which the comparison operator returns false when compare to value. The order of the elements that are not removed is preserved.

Effects: Copies all the elements referred to by the iterator it in the range [first, last) for which the following corresponding conditions do not hold: `*it == value`

The assignments in the parallel `remove_copy` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs not more than `last - first` assignments, exactly `last - first` applications of the predicate *f*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type that the result of dereferencing `FwdIter1` is compared to.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `val`: Value to be removed.

The assignments in the parallel `remove_copy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `remove_copy` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `remove_copy` algorithm returns the iterator to the element past the last element copied.

```
template<typename InIter, typename OutIter, typename Pred>
FwdIter remove_copy_if (InIter first, InIter last, OutIter dest, Pred &&pred)
```

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the predicate *f* returns false. The order of the elements that are not removed is preserved.

Effects: Copies all the elements referred to by the iterator *it* in the range [first, last) for which the following corresponding conditions do not hold: *Invoke*(*pred*, **it*) != false.

The assignments in the parallel *remove_copy_if* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *f*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced).
- *Pred*: The type of the function/function object to use (deduced).

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the elements to be removed. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

Return The *remove_copy_if* algorithm returns an *OutIter*. The *remove_copy_if* algorithm returns the iterator to the element past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred>
FwdIter remove_copy_if (ExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest, Pred &&pred)
```

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the predicate *f* returns false. The order of the elements that are not removed is preserved.

Effects: Copies all the elements referred to by the iterator *it* in the range [first, last) for which the following corresponding conditions do not hold: *Invoke*(*pred*, **it*) != false.

The assignments in the parallel *remove_copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *f*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `remove_copy_if` requires `Pred` to meet the requirements of `CopyConstructible`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the elements to be removed. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`.

The assignments in the parallel `remove_copy_if` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `remove_copy_if` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `remove_copy_if` algorithm returns the iterator to the element past the last element copied.

namespace hpx

Functions

```
template<typename InIter, typename T>
void replace(InIter first, InIter last, T const &old_value, T const &new_value)
    Replaces all elements satisfying specific criteria with new_value in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first, last) with new_value, when the
following corresponding conditions hold: *it == old_value
```

The assignments in the parallel *replace* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `T`: The type of the old and new values to replace (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `old_value`: Refers to the old value of the elements to replace.
- `new_value`: Refers to the new value to use as the replacement.

Return The *replace* algorithm returns a `void`.

```
template<typename ExPolicy, typename FwdIter, typename T>
parallel::util::detail::algorithm_result<ExPolicy, void>::type replace(ExPolicy &&policy, FwdIter
first, FwdIter last, T const
&old_value, T const
&new_value)
```

Replaces all elements satisfying specific criteria with `new_value` in the range `[first, last)`.

Effects: Substitutes elements referred by the iterator `it` in the range `[first, last)` with `new_value`, when the following corresponding conditions hold: `*it == old_value`

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `T`: The type of the old and new values to replace (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `old_value`: Refers to the old value of the elements to replace.
- `new_value`: Refers to the new value to use as the replacement.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

```
template<typename Iter, typename Pred, typename T>
void replace_if(Iter first, Iter last, Pred &&pred, T const &new_value)
    Replaces all elements satisfying specific criteria (for which predicate f returns true) with new_value in the range [first, last).
```

Effects: Substitutes elements referred by the iterator *it* in the range [first, last) with *new_value*, when the following corresponding conditions hold: *(INVOKE(f, *it) != false*

The assignments in the parallel *replace_if* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* applications of the predicate.

Template Parameters

- *Iter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- *T*: The type of the new values to replace (deduced).

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- *new_value*: Refers to the new value to use as the replacement.

Return The *replace_if* algorithm returns *void*.

```
template<typename ExPolicy, typename FwdIter, typename Pred, typename T>
parallel::util::detail::algorithm_result<ExPolicy, void>::type replace_if(ExPolicy      &&policy,
                                                               FwdIter   first,   FwdIter
                                                               last,   Pred   &&pred,   T
                                                               const &new_value)
```

Replaces all elements satisfying specific criteria (for which predicate *f* returns true) with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator *it* in the range [first, last) with new_value, when the following corresponding conditions hold: INVOKE(f, *it) != false

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* applications of the predicate.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- *T*: The type of the new values to replace (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *new_value*: Refers to the new value to use as the replacement.

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_if* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

```
template<typename InIter, typename OutIter, typename T>
OutIter replace_copy(InIter first, InIter last, OutIter dest, T const &old_value, T const
&new_value)
```

Copies the all elements from the range [first, last) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator *it* in the range [result, result + (last - first)) either *new_value* or *(first + (it - result)) depending on whether the following corresponding condition holds: *(first + (i - result)) == *old_value*

The assignments in the parallel *replace_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* applications of the predicate. of the algorithm may be parallelized and the manner in which it executes the assignments.

Template Parameters

- **InIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T**: The type of the old and new values (deduced).

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **old_value**: Refers to the old value of the elements to replace.
- **new_value**: Refers to the new value to use as the replacement.

Return The *replace_copy* algorithm returns an *OutIter*. The *replace_copy* algorithm returns the Iterator to the element past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type replace_copy(ExPolicy    &&policy,
                                                               FwdIter1
                                                               first,      FwdIter1
                                                               last,       FwdIter2
                                                               dest,      T   const
                                                               &old_value,
                                                               T           const
                                                               &new_value)
```

Copies the all elements from the range [first, last) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator *it* in the range [result, result + (last - first)) either *new_value* or *(first + (it - result)) depending on whether the following corresponding condition holds: *(first + (i - result)) == *old_value*

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* applications of the predicate.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the old and new values (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `old_value`: Refers to the old value of the elements to replace.
- `new_value`: Refers to the new value to use as the replacement.

The assignments in the parallel `replace_copy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `replace_copy` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `replace_copy` algorithm returns the Iterator to the element past the last element copied.

```
template<typename InIter, typename OutIter, typename Pred, typename T>
OutIter replace_copy_if(InIter first, InIter last, OutIter dest, Pred &&pred, T const
&new_value)
```

Copies the all elements from the range `[first, last)` to another range beginning at `dest` replacing all elements satisfying a specific criteria with `new_value`.

Effects: Assigns to every iterator it in the range `[result, result + (last - first))` either `new_value` or `*(first + (it - result))` depending on whether the following corresponding condition holds: `INVOKE(f, *(first + (i - result))) != false`

The assignments in the parallel `replace_copy_if` algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` applications of the predicate.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `equal` requires `Pred` to meet the requirements of `CopyConstructible`. (deduced).
- `T`: The type of the new values to replace (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the elements which need to 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 `InIter` can be dereferenced and then implicitly converted to `Type`.

- `new_value`: Refers to the new value to use as the replacement.

Return The `replace_copy_if` algorithm returns an `OutIter`. The `replace_copy_if` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred, typename T>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type replace_copy_if(ExPolicy
    &&policy,
    FwdIter1 first,
    FwdIter1 last,
    FwdIter2 dest,
    Pred &&pred,
    T      const
    &new_value)
```

Copies the all elements from the range [first, last) to another range beginning at `dest` replacing all elements satisfying a specific criteria with `new_value`.

Effects: Assigns to every iterator `it` in the range [result, result + (last - first)) either `new_value` or `*(first + (it - result))` depending on whether the following corresponding condition holds: `INVOKED(f, *(first + (i - result))) != false`

The assignments in the parallel `replace_copy_if` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` applications of the predicate.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `equal` requires `F` to meet the requirements of `CopyConstructible`. (deduced).
- `T`: The type of the new values to replace (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the elements which need to 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.

The assignments in the parallel `replace_copy_if` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `replace_copy_if` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `replace_copy_if` algorithm returns the iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

Functions

```
template<typename BidirIter>
void reverse(BidirIter first, BidirIter last)
```

Reverses the order of the elements in the range [first, last). Behaves as if applying `std::iter_swap` to every pair of iterators `first+i, (last-i) - 1` for each non-negative $i < (last-first)/2$.

The assignments in the parallel `reverse` algorithm execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between `first` and `last`.

Template Parameters

- `BidirIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an bidirectional iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The `reverse` algorithm returns a `void`.

```
template<typename ExPolicy, typename BidirIter>
parallel::util::detail::algorithm_result<ExPolicy, void>::type reverse(ExPolicy &&policy, BidirIter
first, BidirIter last)
```

Reverses the order of the elements in the range [first, last). Behaves as if applying `std::iter_swap` to every pair of iterators `first+i, (last-i) - 1` for each non-negative $i < (last-first)/2$.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **BidirIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an bidirectional iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *reverse* algorithm returns a `hpx::future<void>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

```
template<typename BidirIter, typename OutIter>
OutIter reverse_copy(BidirIter first, BidirIter last, OutIter dest)
```

Copies the elements from the range [first, last) to another range beginning at *dest_first* in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(\text{dest_first} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [first, last) and [*dest_first*, *dest_first*+(*last-first*)) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **BidirIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an bidirectional iterator.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the begin of the destination range.

Return The *reverse_copy* algorithm returns an *OutIter*. The *reverse_copy* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename BidirIter, typename FwdIter>
```

```
util::detail::algorithm_result<ExPolicy, FwdIter>::type reverse_copy (ExPolicy &&policy, BidirIter  
first, BidirIter last, FwdIter  
dest_first)
```

Copies the elements from the range [first, last) to another range beginning at dest_first in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(dest_first + (last - first) - 1 - i) = *(first + i) once for each non-negative i < (last - first). If the source and destination ranges (that is, [first, last) and [dest_first, dest_first+(last-first)) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **BidirIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an bidirectional iterator.
- **FwdIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the begin of the destination range.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *reverse_copy* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *reverse_copy* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

Functions

```
template<typename FwdIter>  
FwdIter rotate (FwdIter first, FwdIter new_first, FwdIter last)
```

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element *new_first* becomes the first element of the new range and *new_first - 1* becomes the last element.

Note Complexity: Linear in the distance between *first* and *last*.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Return The *rotate* algorithm returns a *FwdIter*. The *rotate* algorithm returns the iterator to the new location of the element pointed by *first*, equal to *first + (last - new_first)*.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *new_first*: Refers to the element that should appear at the beginning of the rotated range.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to. The assignments in the parallel *rotate* algorithm execute in sequential order in the calling thread.

```
template<typename ExPolicy, typename FwdIter>
util::detail::algorithm_result<ExPolicy, FwdIter>::type rotate (ExPolicy &&policy, FwdIter first,
                                                               FwdIter new_first, FwdIter last)
```

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [*first*, *last*) in such a way that the element *new_first* becomes the first element of the new range and *new_first - 1* becomes the last element.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *new_first*: Refers to the element that should appear at the beginning of the rotated range.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Return The *rotate* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *rotate* algorithm returns the iterator equal to *first + (last - new_first)*.

```
template<typename FwdIter, typename OutIter>
OutIter rotate_copy (FwdIter first, FwdIter new_first, FwdIter last, OutIter dest_first)
```

Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *new_first* becomes the first element of the new range and *new_first* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **OutIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **new_first**: Refers to the element that should appear at the beginning of the rotated range.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest_first**: Refers to the begin of the destination range.

Return The *rotate_copy* algorithm returns a output iterator, The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type rotate_copy (ExPolicy &&policy, FwdIter1
                                                               first, FwdIter1 new_first,
                                                               FwdIter1 last, FwdIter2
                                                               dest_first)
```

Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *new_first* becomes the first element of the new range and *new_first* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `new_first`: Refers to the element that should appear at the beginning of the rotated range.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest_first`: Refers to the begin of the destination range.

The assignments in the parallel `rotate_copy` algorithm execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `rotate_copy` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `parallel_task_policy` and returns `FwdIter2` otherwise. The `rotate_copy` algorithm returns the output iterator to the element past the last element copied.

namespace hpx

Functions

`template<typename FwdIter, typename FwdIter2, typename Pred = detail::equal_to>`
`FwdIter search(FwdIter first, FwdIter last, FwdIter2 s_first, FwdIter2 s_last, Pred &&op = Pred())`
 Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel `search` algorithm execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_{\text{first}}, s_{\text{last}})$ and $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- `FwdIter`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `adjacent_find` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::equal_to<>`

Parameters

- `first`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `s_first`: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- `s_last`: Refers to the end of the sequence of elements of the algorithm will be searching for.
- `op`: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

<code>bool pred(const Type1 &a, const Type2 &b);</code>

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Return The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

```
template<typename ExPolicy, typename FwdIter, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result<ExPolicy, FwdIter>::type search(ExPolicy &&policy, FwdIter first,
                                                               FwdIter last, FwdIter2 s_first, FwdIter2
                                                               s_last, Pred &&op = Pred())
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_{\text{first}}, s_{\text{last}})$ and $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to<>*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *s_first*: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- *s_last*: Refers to the end of the sequence of elements of the algorithm will be searching for.
- *op*: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

The comparison operations in the parallel `search` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `search` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `task_execution_policy` and returns `FwdIter` otherwise. The `search` algorithm returns an iterator to the beginning of the first subsequence $[s_{\text{first}}, s_{\text{last}}]$ in range $[\text{first}, \text{last}]$. If the length of the subsequence $[s_{\text{first}}, s_{\text{last}}]$ is greater than the length of the range $[\text{first}, \text{last}]$, `last` is returned. Additionally if the size of the subsequence is empty `first` is returned. If no subsequence is found, `last` is returned.

```
template<typename FwdIter, typename FwdIter2, typename Pred = detail::equal_to>
FwdIter search_n(FwdIter first, std::size_t count, FwdIter2 s_first, FwdIter2 s_last, Pred &&op = Pred())
Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.
```

The comparison operations in the parallel `search_n` algorithm execute in sequential order in the calling thread.

Note Complexity: at most $(S*N)$ comparisons where $S = \text{distance}(s_{\text{first}}, s_{\text{last}})$ and $N = \text{count}$.

Template Parameters

- `FwdIter`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `adjacent_find` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::equal_to<>`

Parameters

- `first`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `count`: Refers to the range of elements of the first range the algorithm will be applied to.
- `s_first`: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- `s_last`: Refers to the end of the sequence of elements of the algorithm will be searching for.
- `op`: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

Return The *search_n* algorithm returns *FwdIter*. The *search_n* algorithm returns an iterator to the beginning of the last subsequence [s_first, s_last] in range [first, first+count]. If the length of the subsequence [s_first, s_last] is greater than the length of the range [first, first+count], *first* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *first* is also returned.

```
template<typename ExPolicy, typename FwdIter, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result<ExPolicy, FwdIter>::type search_n(ExPolicy &&policy, FwdIter first,
                                                               std::size_t count, FwdIter2 s_first,
                                                               FwdIter2 s_last, Pred &&op =
                                                               Pred())
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_{\text{first}}, s_{\text{last}})$ and $N = \text{count}$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **count**: Refers to the range of elements of the first range the algorithm will be applied to.
- **s_first**: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last**: Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op**: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

The comparison operations in the parallel *search_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *search_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search_n* algorithm returns an iterator to the beginning of the last subsequence [s_first, s_last) in range [first, first+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.

namespace hpx

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = detail::util::detail::algorithm_result<ExPolicy, FwdIter3>::type::type set_difference(ExPolicy &&policy,
    FwdIter1 first1,
    FwdIter1 last1,
    FwdIter2 first2,
    FwdIter2 last2,
    FwdIter3 dest, Pred
    &&op = Pred())
```

Constructs a sorted range beginning at *dest* consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *FwdIter1*: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the source iterators used (deduced) representing the second 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<*T*>

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.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = detail::util::detail::algorithm_result<ExPolicy, FwdIter3>::type set_intersection(ExPolicy &&policy,
FwdIter1 first1,
FwdIter1 last1,
FwdIter2 first2,
FwdIter2 last2,
FwdIter3 dest, Pred
&&op = Pred())
```

Constructs a sorted range beginning at `dest` consisting of all elements present in both sorted ranges `[first1, last1]` and `[first2, last2]`. This algorithm expects both input ranges to be sorted with the given binary predicate `f`.

If some element is found m times in $[first1, last1]$ and n times in $[first2, last2]$, the first $\min(m, n)$ 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.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = detail::util::detail::algorithm_result<ExPolicy, FwdIter3>::type::type set_symmetric_difference(ExPolicy &&policy, FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2, FwdIter3 dest, Pred &&op = Pred())
```

Constructs a sorted range beginning at *dest* consisting of all elements present in either of the sorted ranges [*first1*, *last1*] and [*first2*, *last2*], but not in both of them are copied to the range beginning at *dest*. The resulting range is also sorted. This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [*first1*, *last1*] and *n* times in [*first2*, *last2*], it will be copied to *dest* exactly `std::abs(m-n)` times. If *m*>*n*, then the last *m-n* of those elements are copied from [*first1*,*last1*), otherwise the last *n-m* elements are copied from [*first2*,*last2*). The resulting range cannot overlap with either of the input ranges.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *FwdIter1*: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- *FwdIter3*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last1*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *first2*: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- *last2*: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *set_symmetric_difference* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = detail::util::detail::algorithm_result<ExPolicy, FwdIter3>::type::type set_union(ExPolicy &&policy,
    FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2, FwdIter3 dest, Pred &&op = Pred())
```

Constructs a sorted range beginning at *dest* consisting of all elements present in one or both sorted ranges [*first1*, *last1*) and [*first2*, *last2*). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [*first1*, *last1*) and *n* times in [*first2*, *last2*), then all *m* elements will be copied from [*first1*, *last1*) to *dest*, preserving order, and then exactly std::max(*n-m*, 0) elements will be copied from [*first2*, *last2*) to *dest*, also preserving order.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where $N1$ is the length of the first sequence and $N2$ is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `FwdIter1`: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- `FwdIter3`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Op`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `set_union` requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `set_union` algorithm returns a `hpx::future<FwdIter3>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter3` otherwise. The `set_union` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Size>
FwdIter shift_left (FwdIter first, FwdIter last, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position `first + n + i` to position `first + i`.

The assignment operations in the parallel `shift_left` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Size`: The type of the argument specifying the number of positions to shift by.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `n`: Refers to the number of positions to shift.

Note The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

Return The `shift_left` algorithm returns `FwdIter`. The `shift_left` algorithm returns an iterator to the end of the resulting range.

```
template<typename ExPolicy, typename FwdIter, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter> shift_left (ExPolicy &&policy, FwdIter
first, FwdIter last, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position `first + n + i` to position `first + i`.

The assignment operations in the parallel `shift_left` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of positions to shift by.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **n**: Refers to the number of positions to shift.

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Return The *shift_left* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *shift_left* algorithm returns an iterator to the end of the resulting range.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Size>
FwdIter shift_right(FwdIter first, FwdIter last, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- $n + i$.

The assignment operations in the parallel *shift_right* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of positions to shift by.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **n**: Refers to the number of positions to shift.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Return The *shift_right* algorithm returns *FwdIter*. The *shift_right* algorithm returns an iterator to the end of the resulting range.

```
template<typename ExPolicy, typename FwdIter, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter> shift_right (ExPolicy && policy, FwdIter
```

first, FwdIter last, Size n)
Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), move the element originally at position first + i to position first + i.

The assignment operations in the parallel `shift_right` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: At most (last - first) n assignments

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
 - **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
 - **Size**: The type of the argument specifying the number of positions to shift by.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
 - **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
 - **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
 - **n**: Refers to the number of positions to shift.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Return The `shift_right` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. The `shift_right` algorithm returns an iterator to the end of the resulting range.

namespace hpx

Functions

```
template<typename RandomIt>
void sort (RandomIt first, RandomIt last)
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i + n*)), INVOKE(*proj*, **i*)) == false.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Template Parameters

- RandomIt: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.

Parameters

- first: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- last: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The *sort* algorithm does not return anything.

```
template<typename ExPolicy, typename RandomIt>
util::detail::algorithm_result<ExPolicy>::type sort (ExPolicy &&policy, RandomIt first, RandomIt last)
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i + n*)), INVOKE(*proj*, **i*)) == false.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- ExPolicy: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- RandomIt: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *sort* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

```
template<typename RandomIt, typename Comp, typename Proj>
void sort (RandomIt first, RandomIt last, Comp &&comp, Proj &&proj)
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i + n*)), INVOKE(*proj*, **i*)) == false.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

comp has to induce a strict weak ordering on the values.

Template Parameters

- **RandomIt**: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp**: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Return The *sort* algorithm returns nothing.

```
template<typename ExPolicy, typename RandomIt, typename Comp, typename Proj>
```

```
parallel::util::detail::algorithm_result<ExPolicy>::type sort (ExPolicy &&policy, RandomIt first, RandomIt last, Comp &&comp, Proj &&proj)
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

comp has to induce a strict weak ordering on the values.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **RandomIt**: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp**: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type **Comp**, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *sort* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

```
namespace hpx
```

```
namespace parallel
```

Typedefs

```
template<typename KeyIter, typename ValueIter>
using sort_by_key_result = std::pair<KeyIter, ValueIter>
```

Functions

```
template<typename ExPolicy, typename KeyIter, typename ValueIter, typename Compare = detail::less>
util::detail::algorithm_result_t<ExPolicy, sort_by_key_result<KeyIter, ValueIter>> sort_by_key(ExPolicy
    &&policy,
    KeyIter
    key_first,
    KeyIter
    key_last,
    Val-
    ueIter
    value_first,
    Com-
    pare
    &&comp
    =
    Com-
    pare())
=
```

Sorts one range of data using keys supplied in another range. The key elements in the range [key_first, key_last) are sorted in ascending order with the corresponding elements in the value range moved to follow the sorted order. The algorithm is not stable, the order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i + n*)), INVOKE(*proj*, **i*)) == false.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

comp has to induce a strict weak ordering on the values.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **KeyIter**: The type of the key iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **ValueIter**: The type of the value iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **key_first**: Refers to the beginning of the sequence of key elements the algorithm will be applied to.
- **key_last**: Refers to the end of the sequence of key elements the algorithm will be applied to.

- `value_first`: Refers to the beginning of the sequence of value elements the algorithm will be applied to, the range of elements must match `[key_first, key_last]`
- `comp`: `comp` is a callable object. The return value of the `INVOKE` operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

The application of function objects in parallel algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `sort_by-key` algorithm returns a `hpx::future<sort_by_key_result<KeyIter, ValueIter>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns *otherwise*. The algorithm returns a pair holding an iterator pointing to the first element after the last element in the input key sequence and an iterator pointing to the first element after the last element in the input value sequence.

```
namespace hpx
```

Functions

```
template<typename RandomIt>
void stable_sort (RandomIt first, RandomIt last)
```

Sorts the elements in the range `[first, last)` in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object `comp` (defaults to using `operator<()`).

A sequence is sorted with respect to a comparator `comp` and a projection `proj` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, and `INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false`.

Note Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ comparisons.

The assignments in the parallel `stable_sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Template Parameters

- `RandomIt`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The `stable_sort` algorithm does not return anything.

```
template<typename ExPolicy, typename RandomIt>
```

util::detail::algorithm_result<ExPolicy>::type **stable_sort** (*ExPolicy &&policy, RandomIt first, RandomIt last*)

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i + n*)), INVOKE(*proj*, **i*)) == false.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *RandomIt*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *stable_sort* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

template<typename **RandomIt**, typename **Comp**, typename **Proj**>

void **stable_sort** (*RandomIt first, RandomIt last, Comp &&comp, Proj &&proj*)

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i + n*)), INVOKE(*proj*, **i*)) == false.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

comp has to induce a strict weak ordering on the values.

Template Parameters

- *RandomIt*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- *Comp*: The type of the function/function object to use (deduced).

- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp**: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.
- **proj**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.

The assignments in the parallel `stable_sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Return The `stable_sort` algorithm returns nothing.

```
template<typename ExPolicy, typename RandomIt, typename Comp, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy>::type stable_sort (ExPolicy &&policy, RandomIt
                                                               first, RandomIt last, Comp
                                                               &&comp, Proj &&proj)
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object `comp` (defaults to using operator`<`).

A sequence is sorted with respect to a comparator `comp` and a projection `proj` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, and `INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false`.

Note Complexity: $O(N \log(N))$, where $N = \text{std}::distance(first, last)$ comparisons.

`comp` has to induce a strict weak ordering on the values.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **RandomIt**: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp**: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

- `proj`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `stable_sort` algorithm returns a `hpx::future<void>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns nothing otherwise.

```
namespace hpx
```

Variables

```
hpx::starts_with_t starts_with
struct starts_with_t : public hpx::functional::detail::tagFallback<starts_with_t>
```

Friends

```
template<typename InIter1, typename InIter2, typename Pred = hpx::parallel::v1::detail::equal_to, typename Proj = ranges::equal_to<ExPolicy>, typename FwdIter1, typename FwdIter2, typename Pred = ranges::equal_to, typename Proj = ranges::equal_to>
bool tagFallback_invoke(hpx::starts_with_t, InIter1 first1, InIter1 last1, InIter2 first2,
                        InIter2 last2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(),
                        Proj2 &&proj2 = Proj2())

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = ranges::equal_to, typename Proj = ranges::equal_to>
type tagFallback_invoke(hpx::starts_with_t,
                        ExPolicy
                        &&pol-
                        icy,
                        FwdIter1
                        first1,
                        FwdIter1
                        last1,
                        FwdIter2
                        first2,
                        FwdIter2
                        last2,
                        Pred
                        &&pred
                        = Pred(),
                        Proj1
                        &&proj1
                        = Proj1(),
                        Proj2
                        &&proj2
                        =
                        Proj2())
```

```
namespace hpx
```

Functions

```
template<typename FwdIter1, typename FwdIter2>
FwdIter2 swap_ranges(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2)
    Exchanges elements between range [first1, last1) and another range starting at first2.
```

The swap operations in the parallel *swap_ranges* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first1* and *last1*

Template Parameters

- *FwdIter1*: The type of the first range of iterators to swap (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the second range of iterators to swap (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *first1*: Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- *last1*: Refers to the end of the first sequence of elements the algorithm will be applied to.
- *first2*: Refers to the beginning of the second sequence of elements the algorithm will be applied to.

Return The *swap_ranges* algorithm returns *FwdIter2*. The *swap_ranges* algorithm returns iterator to the element past the last element exchanged in the range beginning with *first2*.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type swap_ranges(ExPolicy &&policy,
    FwdIter1 first1,
    FwdIter1 last1,
    FwdIter2 first2)
```

Exchanges elements between range [first1, last1) and another range starting at first2.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first1* and *last1*

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the swap operations.
- *FwdIter1*: The type of the first range of iterators to swap (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the second range of iterators to swap (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.

- `first1`: Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- `last1`: Refers to the end of the first sequence of elements the algorithm will be applied to.
- `first2`: Refers to the beginning of the second sequence of elements the algorithm will be applied to.

The swap operations in the parallel `swap_ranges` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `swap_ranges` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `parallel_task_policy` and returns `FwdIter2` otherwise. The `swap_ranges` algorithm returns iterator to the element past the last element exchanged in the range beginning with `first2`.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename F>
util::detail::algorithm_result<ExPolicy, util::in_out_result<FwdIter1, FwdIter2>>::type transform(ExPolicy
    &&policy,
    FwdIter1
    first,
    FwdIter1
    last,
    FwdIter2
    dest,
    F
    &&f)
```

Applies the given function f to the range $[first, last)$ and stores the result in another range, beginning at `dest`.

The invocations of f in the parallel `transform` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Exactly $last - first$ applications of f

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of f .
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `transform` requires `F` to meet the requirements of `CopyConstructible`.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

The signature does not need to have `const&`. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `Ret` must be such that an object of type `FwdIter2` can be dereferenced and assigned a value of type `Ret`.

The invocations of `f` in the parallel `transform` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `transform` algorithm returns a `hpx::future<in_out_result<FwdIter1, FwdIter2>>` if the execution policy is of type `parallel_task_policy` and returns `in_out_result<FwdIter1, FwdIter2>` otherwise. The `transform` algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename F>
util::detail::algorithm_result<ExPolicy, util::in_in_out_result<FwdIter1, FwdIter2, FwdIter3>>::type transform(ExPolicy
&&pol-
icy,
FwdIter1,
first1,
FwdIter1,
last1,
FwdIter2,
first2,
FwdIter3,
dest,
F
&&f)
```

Applies the given function `f` to pairs of elements from two ranges: one defined by `[first1, last1)` and the other beginning at `first2`, and stores the result in another range, beginning at `dest`.

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*.

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*.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *transform* algorithm returns a `hpx::future<in_in_out_result<FwdIter1, FwdIter2, FwdIter3>>` if the execution policy is of type *parallel_task_policy* and returns `in_in_out_result<FwdIter1, FwdIter2, FwdIter3>` otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

namespace hpx

Functions

```
template<typename InIter, typename OutIter, typename T, typename BinOp, typename UnOp>
OutIter transform_exclusive_scan(InIter first, InIter last, OutIter dest, T init, BinOp &&binary_op, UnOp &&unary_op)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
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.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

Return The *transform_exclusive_scan* algorithm returns a returns *OutIter*. The *transform_exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ when N is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*_M, ..., *a*_N) where 1 < K+1 = M <= N).

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename BinOp, typename UnOp>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type transform_exclusive_scan(ExPolicy
                                         &&policy,
                                         FwdIter1
                                         first,
                                         FwdIter1
                                         last,
                                         FwdIter2
                                         dest,
                                         T
                                         init,
                                         BinOp
                                         &&bi-
                                         nary_op,
                                         UnOp
                                         &&unary_op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: O(*last - first*) applications of the predicates *op* and *conv*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `conv`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

- `init`: The initial value for the generalized sum.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel `transform_exclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The `transform_exclusive_scan` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `transform_exclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- a_1 when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_M, \dots, a_N))$ where $1 < K+1 = M \leq N$.

The behavior of `transform_exclusive_scan` may be non-deterministic for a non-associative predicate.

```
namespace hpx
```

Functions

```
template<typename InIter, typename OutIter, typename BinOp, typename UnOp>
OutIter transform_inclusive_scan(InIter first, InIter last, OutIter dest, BinOp &&binary_op,
                                  UnOp &&unary_op)
Assigns through each iterator i in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), ..., conv(*(first + (i - result)))).
```

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [*first*, *last*). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [*first*,*last*) or [result,result + (last - first)).

Return The *transform_inclusive_scan* algorithm returns a returns *OutIter*. The *transform_inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ when N is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*_M, ..., *a*_N)) where 1 < K+1 = M <= N.

The difference between *inclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the *i*th input element in the *i*th sum.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename BinOp, typename UnOp>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type transform_inclusive_scan(ExPolicy
    &&policy,
    FwdIter1
    first,
    FwdIter1
    last,
    FwdIter2
    dest,
    BinOp
    &&binary_op,
    UnOp
    &&unary_op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(*op*, conv(*first), ..., conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: O(*last - first*) applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Conv**: The type of the unary function object used for the conversion operation.
- **Op**: The type of the binary function object used for the reduction operation.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence 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.

- 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`.

The reduce operations in the parallel `transform_inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The `transform_inclusive_scan` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `transform_inclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- $a1$ when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \dots, aN))$ where $1 < K+1 = M \leq N$.

The difference between `inclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the i th input element in the i th sum.

```
template<typename InIter, typename OutIter, typename BinOp, typename UnOp, typename T>
OutIter transform_inclusive_scan(InIter first, InIter last, OutIter dest, BinOp &&binary_op,
                               UnOp &&unary_op, T init)
Assigns through each iterator  $i$  in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, conv(*first), \dots, conv(*(first + (i - result))))).
```

The reduce operations in the parallel `transform_inclusive_scan` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `op`.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Conv`: The type of the unary function object used for the conversion operation.
- `Op`: The type of the binary function object used for the reduction operation.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- `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.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

Return The `transform_inclusive_scan` algorithm returns a returns `OutIter`. The `transform_inclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- `a1` when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \dots, aN))$ where $1 < K+1 = M \leq N$.

The difference between `inclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the i th input element in the i th sum. `op` is not mathematically associative, the behavior of `transform_inclusive_scan` may be non-deterministic.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **BinOp**, typename **UnOp**, typename **T**

```
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type transform_inclusive_scan(ExPolicy
    &&policy,
    FwdIter1
    first,
    FwdIter1
    last,
    FwdIter2
    dest,
    BinOp
    &&binary_op,
    UnOp
    &&unary_op,
    T
    init)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM($op, init, conv(*first), \dots, conv(*(first + (i - result)))$)).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate op .

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *Op*: The type of the binary function object used for the reduction operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- `conv`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by `[first, last)`. This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

- `init`: The initial value for the generalized sum.

The reduce operations in the parallel `transform_inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

Return The `transform_inclusive_scan` algorithm returns a `hpx::future<FwdIter2>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `transform_inclusive_scan` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- a_1 when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_M, \dots, a_N))$ where $1 < K+1 = M \leq N$.

The difference between `inclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the i th input element in the i th sum. `op` is not mathematically associative, the behavior of `transform_inclusive_scan` may be non-deterministic.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter, typename T, typename Reduce, typename Convert>
util::detail::algorithm_result<ExPolicy, T>::type transform_reduce (ExPolicy &&policy, FwdIter
first, FwdIter last, T init,
Reduce &&red_op, Convert
&&conv_op)
```

Returns `GENERALIZED_SUM(red_op, init, conv_op(*first), ..., conv_op(*(first + (last - first) - 1)))`.

The reduce operations in the parallel `transform_reduce` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates `red_op` and `conv_op`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `copy_if` requires `F` to meet the requirements of `CopyConstructible`.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).
- `Reduce`: The type of the binary function object used for the reduction operation.
- `Convert`: The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `conv_op`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

- `init`: The initial value for the generalized sum.
- `red_op`: Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of `conv_op`. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1`, `Type2`, and `Ret` must be such that an object of a type as returned from `conv_op` can be implicitly converted to any of those types.

The reduce operations in the parallel `transform_reduce` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `transform_reduce` and `accumulate` is that the behavior of `transform_reduce` may be non-deterministic for non-associative or non-commutative binary predicate.

Return The `transform_reduce` algorithm returns a `hpx::future<T>` if the execution policy is of type `parallel_task_policy` and returns `T` otherwise. The `transform_reduce` algorithm returns the result of the generalized sum over the values returned from `conv_op` when applied to the elements given by the input range [first, last).

Note `GENERALIZED_SUM(op, a1, ..., aN)` is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, ..., bK), GENERALIZED_SUM(op, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - $1 < K+1 = M \leq N$.

```
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: $O(last - first)$ 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 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, Reduce
                                                               &&red_op, Convert
                                                               &&conv_op)
```

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op2*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the first source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *T*: The type of the value to be used as return) values (deduced).
- *Reduce*: The type of the binary function object used for the multiplication operation.
- *Convert*: The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the first sequence of elements the result will be calculated with.
- *last1*: Refers to the end of the first sequence of elements the algorithm will be applied to.
- *first2*: Refers to the beginning of the second sequence of elements the result will be calculated with.
- *init*: The initial value for the sum.
- *red_op*: Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *op2*. This is a binary predicate. The signature of this predicate should be equivalent to:

```
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.

```
namespace hpx
```

Functions

```
template<typename InIter, typename FwdIter>
FwdIter uninitialized_copy (InIter first, InIter last, FwdIter dest)
```

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *FwdIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.

Return The *uninitialized_copy* algorithm returns *FwdIter*. The *uninitialized_copy* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type uninitialized_copy (ExPolicy
&&policy,
FwdIter1
first,
FwdIter1
last,
FwdIter2
dest)
```

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The assignments in the parallel `uninitialized_copy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `uninitialized_copy` algorithm returns a `hpx::future<FwdIter2>`, if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `uninitialized_copy` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename InIter, typename Size, typename FwdIter>
FwdIter uninitialized_copy_n(InIter first, Size count, FwdIter dest)
```

Copies the elements in the range $[first, first + count]$, starting from `first` and proceeding to `first + count - 1`, to another range beginning at `dest`. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel `uninitialized_copy_n` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` assignments, if `count > 0`, no assignments otherwise.

Template Parameters

- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `Size`: The type of the argument specifying the number of elements to apply f to.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at `first` the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

Return The *uninitialized_copy_n* algorithm returns a returns *FwdIter2*. The *uninitialized_copy_n* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type uninitialized_copy_n(ExPolicy
&&policy,
FwdIter1
first,
Size
count,
FwdIter2
dest)
```

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count* > 0, no assignments otherwise.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Size*: The type of the argument specifying the number of elements to apply *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.

namespace hpx

Functions

```
template<typename FwdIter>
void uninitialized_default_construct (FwdIter first, FwdIter last)
    Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage
    designated by the range by default-initialization. If an exception is thrown during the initialization, the
    function has no effects.
```

The assignments in the parallel *uninitialized_default_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The *uninitialized_default_construct* algorithm returns nothing

```
template<typename ExPolicy, typename FwdIter>
parallel::util::detail::algorithm_result<ExPolicy>::type uninitialized_default_construct (ExPolicy
    &&policy,
    FwdIter
    first,
    FwdIter
    last)
```

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel `uninitialized_default_construct` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `uninitialized_default_construct` algorithm returns a `hpx::future<void>`, if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns nothing otherwise.

```
template<typename FwdIter, typename Size>
FwdIter uninitialized_default_construct_n(FwdIter first, Size count)
```

Constructs objects of type typename iterator_traits<`ForwardIt`>::value_type in the uninitialized storage designated by the range [first, first + count) by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel `uninitialized_default_construct_n` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` assignments, if `count > 0`, no assignments otherwise.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Size`: The type of the argument specifying the number of elements to apply `f` to.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at `first` the algorithm will be applied to.

Return The `uninitialized_default_construct_n` algorithm returns a returns `FwdIter`. The `uninitialized_default_construct_n` algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename ExPolicy, typename FwdIter, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_default_construct_n(ExPolicy
&&policy,
FwdIter
first,
Size
count)
```

Constructs objects of type typename iterator_traits<`ForwardIt`>::value_type in the uninitialized storage designated by the range [first, first + count) by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel `uninitialized_default_construct_n` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` assignments, if `count > 0`, no assignments otherwise.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of elements to apply *f* to.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count**: Refers to the number of elements starting at *first* the algorithm will be applied to.

The assignments in the parallel *uninitialized_default_construct_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_default_construct_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_default_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename T>
void uninitialized_fill(FwdIter first, FwdIter last, T const &value)
    Copies the given value to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.
```

The assignments in the parallel *uninitialized_fill* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be assigned (deduced).

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **value**: The value to be assigned.

Return The *uninitialized_fill* algorithm returns nothing

```
template<typename ExPolicy, typename FwdIter, typename T>
```

```
parallel::util::detail::algorithm_result<ExPolicy>::type uninitialized_fill (ExPolicy    &&policy,
                                                               FwdIter first,
                                                               FwdIter last, T
                                                               const &value)
```

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

The initializations in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *T*: The type of the value to be assigned (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *value*: The value to be assigned.

The initializations in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_fill* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

```
template<typename FwdIter, typename Size, typename T>
FwdIter uninitialized_n (FwdIter first, Size count, T const &value)
```

Copies the given *value* value to the first *count* elements in an uninitialized memory area beginning at *first*. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill_n* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count* > 0, no assignments otherwise.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *T*: The type of the value to be assigned (deduced).

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *value*: The value to be assigned.

Return The *uninitialized_fill_n* algorithm returns a returns *FwdIter*. The *uninitialized_fill_n* algorithm returns the output iterator to the element in the range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename T>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_fill_n(ExPolicy
    &&policy,
    FwdIter
    first, Size
    count, T
const
&value)
```

Copies the given *value* value to the first *count* elements in an uninitialized memory area beginning at *first*. If an exception is thrown during the initialization, the function has no effects.

The initializations in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count* > 0, no assignments otherwise.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *T*: The type of the value to be assigned (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *value*: The value to be assigned.

The initializations in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_fill_n* algorithm returns a *hpx::future<FwdIter>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_fill_n* algorithm returns the output iterator to the element in the range, one past the last element copied.

namespace hpx

Functions

```
template<typename InIter, typename FwdIter>
FwdIter uninitialized_move(InIter first, InIter last, FwdIter dest)
```

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *FwdIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.

Return The *uninitialized_move* algorithm returns *FwdIter*. The *uninitialized_move* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type uninitialized_move(ExPolicy
&&policy,
FwdIter1
first,
FwdIter1
last,
FwdIter2
dest)
```

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The assignments in the parallel `uninitialized_move` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `uninitialized_move` algorithm returns a `hpx::future<FwdIter2>`, if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter2` otherwise. The `uninitialized_move` algorithm returns the output iterator to the element in the destination range, one past the last element moved.

```
template<typename InIter, typename Size, typename FwdIter>
FwdIter uninitialized_move_n(InIter first, Size count, FwdIter dest)
```

Moves the elements in the range $[first, first + count]$, starting from `first` and proceeding to $first + count - 1$, to another range beginning at `dest`. If an exception is thrown during the initialization, some objects in $[first, first + count)$ are left in a valid but unspecified state.

The assignments in the parallel `uninitialized_move_n` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` movements, if `count > 0`, no move operations otherwise.

Template Parameters

- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `Size`: The type of the argument specifying the number of elements to apply f to.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at `first` the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

Return The `uninitialized_move_n` algorithm returns a returns `FwdIter2`. The `uninitialized_move_n` algorithm returns the output iterator to the element in the destination range, one past the last element moved.

```
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2>
```

```
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type uninitialized_move_n(ExPolicy  
    &&pol-  
    icy,  
    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 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_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<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_move_n* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

namespace hpx

Functions

```
template<typename FwdIter>
void uninitialized_value_construct (FwdIter first, FwdIter last)
    Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage
    designated by the range by value-initialization. If an exception is thrown during the initialization, the
    function has no effects.
```

The assignments in the parallel *uninitialized_value_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The *uninitialized_value_construct* algorithm returns nothing

```
template<typename ExPolicy, typename FwdIter>
parallel::util::detail::algorithm_result<ExPolicy>::type uninitialized_value_construct (ExPolicy
    &&policy,
    FwdIter
    first,
    FwdIter
    last)
```

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel `uninitialized_value_construct` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `uninitialized_value_construct` algorithm returns a `hpx::future<void>`, if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns nothing otherwise.

```
template<typename FwdIter, typename Size>
FwdIter uninitialized_value_construct_n(FwdIter first, Size count)
```

Constructs objects of type typename iterator_traits<`ForwardIt`>::value_type in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel `uninitialized_value_construct_n` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` assignments, if `count > 0`, no assignments otherwise.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Size`: The type of the argument specifying the number of elements to apply `f` to.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at `first` the algorithm will be applied to.

Return The `uninitialized_value_construct_n` algorithm returns a `FwdIter`. The `uninitialized_value_construct_n` algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename ExPolicy, typename FwdIter, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_value_construct_n(ExPolicy
&&policy,
FwdIter
first,
Size
count)
```

Constructs objects of type typename iterator_traits<`ForwardIt`>::value_type in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel `uninitialized_value_construct_n` algorithm invoked with an execution policy object of type `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.

```
namespace hpx
```

Functions

```
template<typename FwdIter>
FwdIter unique (FwdIter first, FwdIter last)
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The *unique* algorithm returns *FwdIter*. The *unique* algorithm returns the iterator to the new end of the range.

```
template<typename ExPolicy, typename FwdIter>
```

`parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type unique (ExPolicy &&policy, FwdIter first, FwdIter last)`

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *unique* algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *unique* algorithm returns the iterator to the new end of the range.

`template<typename ExPolicy, typename FwdIter, typename Pred, typename Proj>`
`FwdIter unique (FwdIter first, FwdIter last, Pred &&pred, Proj &&proj)`

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first - 1* applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *unique* algorithm returns *FwdIter*. The *unique* algorithm returns the iterator to the new end of the range.

```
template<typename ExPolicy, typename FwdIter, typename Pred, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type unique(ExPolicy &&policy, FwdIter
first, FwdIter last, Pred
&&pred, Proj &&proj)
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first - 1* applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter` can be dereferenced and then implicitly converted to both `Type1` and `Type2`

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *unique* algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. The *unique* algorithm returns the iterator to the new end of the range.

```
template<typename InIter, typename OutIter>
OutIter unique_copy(InIter first, InIter last, OutIter dest)
```

Copies the elements from the range `[first, last)`, to another range beginning at `dest` in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs not more than `last - first` assignments.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

Return The *unique_copy* algorithm returns a returns `OutIter`. The *unique_copy* algorithm returns the destination iterator to the end of the `dest` range.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type unique_copy(ExPolicy    &&policy,
                                                               FwdIter1 first,
                                                               FwdIter1 last,
                                                               FwdIter2 dest)
```

Copies the elements from the range `[first, last)`, to another range beginning at `dest` in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *unique_copy* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

```
template<typename InIter, typename OutIter, typename Pred, typename Proj>
OutIter unique_copy(InIter first, InIter last, OutIter dest, Pred &&pred, Proj &&proj)
```

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first - 1* applications of the predicate *pred* and no more than twice as many applications of the projection *proj*

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to<>*
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is 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.

Return The *unique_copy* algorithm returns a returns *OutIter*. The *unique_copy* algorithm returns the destination iterator to the end of the *dest* range.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type unique_copy(ExPolicy      &&policy,
                                                               FwdIter1    first,
                                                               FwdIter1    last,
                                                               FwdIter2    dest,
                                                               Pred &&pred, Proj
                                                               &&proj)
```

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first - 1* applications of the predicate *pred* and no more than twice as many applications of the projection *proj*

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence 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<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

namespace hpx

namespace ranges

Functions

```
template<typename FwdIter, typename Sent, typename Proj = hpx::parallel::util::projection_identity, typename Pred = hpx::parallel::util::less>
FwdIter adjacent_difference(FwdIter first, Sent last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Searches the range [first, last) for two consecutive identical elements.

Note Complexity: Exactly the smaller of (*result* - *first*) + 1 and (*last* - *first*) - 1 application of the predicate where *result* is the value returned

Return The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

Template Parameters

- **FwdIter**: The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*
- **Pred**: The type of an optional function/function object to use.

Parameters

- **first**: Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- `pred`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Proj = hpx::parallel::util::projection_identity>
util::detail::algorithm_result<ExPolicy, FwdIter>::type adjacent_find(ExPolicy policy,
                                                                     FwdIter first, Sent last,
                                                                     Pred pred = Pred(),
                                                                     Proj proj = Proj())
```

Searches the range [first, last) for two consecutive identical elements. This version uses the given binary predicate `pred`

The comparison operations in the parallel `adjacent_find` invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Exactly the smaller of (`result - first`) + 1 and (`last - first`) - 1 application of the predicate where `result` is the value returned

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `adjacent_find` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::equal_to<>`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- `pred`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *adjacent_find* is available if the user decides to provide their algorithm their own binary predicate *pred*.

Return The *adjacent_find* algorithm returns a *hpx::future<InIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *InIter* otherwise. The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

```
template<typename Rng, typename Proj = hpx::parallel::util::projection_identity, typename Pred = detail::equal_to>
hpx::traits::range_traits<Rng>::iterator_type adjacent_difference(ExPolicy &&policy, Rng
&&rng, Pred &&pred =
Pred(), Proj &&proj =
Proj())
```

Searches the range *rng* for two consecutive identical elements.

Note Complexity: Exactly the smaller of (*result* - std::begin(*rng*)) + 1 and (std::begin(*rng*) - std::end(*rng*)) - 1 applications of the predicate where *result* is the value returned

Return The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*
- *Pred*: The type of an optional function/function object to use.

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *pred*: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
template<typename ExPolicy, typename Rng, typename Proj = hpx::parallel::util::projection_identity, typename Pred =
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type adjacent_find
```

Searches the range `rng` for two consecutive identical elements.

The comparison operations in the parallel `adjacent_find` invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Exactly the smaller of $(\text{result} - \text{std::begin}(\text{rng})) + 1$ and $(\text{std::begin}(\text{rng}) - \text{std::end}(\text{rng})) - 1$ applications of the predicate where `result` is the value returned

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). `Thhpx::traits::is_range<Rng>::valuee` iterators extracted from this range type must meet the requirements of an forward iterator.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `adjacent_find` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::equal_to<>`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `pred`: The binary predicate which returns `true` if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel `adjacent_find` invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of `adjacent_find` is available if the user decides to provide their algorithm their own binary predicate `pred`.

Return The `adjacent_find` algorithm returns a `hpx::future<InIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `InIter` otherwise. The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is returned.

Searches the range `rng` for two consecutive identical elements.

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 $(\text{result} - \text{std::begin}(\text{rng})) + 1$ and $(\text{std::begin}(\text{rng}) - \text{std::end}(\text{rng})) - 1$ applications of the predicate where `result` is the value returned

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `adjacent_find` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::equal_to<>`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `pred`: The binary predicate which returns `true` if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel `adjacent_find` invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of `adjacent_find` is available if the user decides to provide their algorithm their own binary predicate `pred`.

Return The `adjacent_find` algorithm returns a `hpx::future<InIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `InIter` otherwise. The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is returned.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter, typename Sent, typename Proj = hpx::parallel::util::projection_identity, typename Pred = hpx::parallel::util::equal_to<>>
FwdIter adjacent_find(FwdIter first, Sent last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Searches the range [first, last) for two consecutive identical elements.

Note Complexity: Exactly the smaller of (`result - first`) + 1 and (`last - first`) - 1 application of the predicate where `result` is the value returned

Return The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is returned.

Template Parameters

- `FwdIter`: The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`
- `Pred`: The type of an optional function/function object to use.

Parameters

- `first`: Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- `pred`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
template<typename Rng, typename Proj = hpx::parallel::util::projection_identity, typename Pred = detail::equal_to>
hpx::traits::range_traits<Rng>::iterator_type adjacent_find(ExPolicy &&policy, Rng &&rng,
    Pred &&pred = Pred(), Proj
    &&proj = Proj())
```

Searches the range `rng` for two consecutive identical elements.

Note Complexity: Exactly the smaller of `(result - std::begin(rng)) + 1` and `(std::begin(rng) - std::end(rng)) - 1` applications of the predicate where `result` is the value returned

Return The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is returned.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`
- `Pred`: The type of an optional function/function object to use.

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `pred`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type none_of(ExPolicy &&policy, Rng &&rng, F
&&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for no elements in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *none_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

```
template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type any_of(ExPolicy &&policy, Rng &&rng, F
&&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies

user-provided function objects.

- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *any_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

```
template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type all_of(ExPolicy &&policy, Rng &&rng, F
&&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for all elements in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most `std::distance(begin(rng), end(rng))` applications of the predicate *f*

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *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 `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.

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename FwdIter>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_result<Iter1, Iter>>::type copy (ExPolicy
&&policy,
Iter1
iter,
Sent1
sent,
FwdIter
dest)
```

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*.

The assignments in the parallel `copy` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Iter1`: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent1`: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for `Iter1`.
- `FwdIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `iter`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `sent`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *copy* algorithm returns a *hpx::future<ranges::copy_result<FwdIter1, FwdIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_result<FwdIter1, FwdIter>* otherwise. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng, typename FwdIter>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_result<typename hpx::traits::range_traits<Rng>::iterator_t<Rng>, FwdIter>> copy(hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_result<typename hpx::traits::range_traits<Rng>::iterator_t<Rng>, FwdIter> const& result);
```

Copies the elements in the range *rng* to another range beginning at *dest*.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng:** The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter:** The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **rng:** Refers to the sequence of elements the algorithm will be applied to.
- **dest:** Refers to the beginning of the destination range.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *copy* algorithm returns a *hpx::future<ranges::copy_result<iterator_t<Rng>, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_result<iterator_t<Rng>, FwdIter2>* otherwise. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2>
hpx::future<ranges::copy_result<iterator_t<Size>, FwdIter2>> copy(hpx::parallel::util::detail::algorithm_result<ExPolicy, ranges::copy_result<iterator_t<Size>, FwdIter2> const& result);
```

hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_n_result<FwdIter1, FwdIter2>>::type **copy_n** (*E*

&
ic
F
fi
S
cc
F
d

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest.

The assignments in the parallel *copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count* > 0, no assignments otherwise.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count**: Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.

The assignments in the parallel *copy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *copy_n* algorithm returns a *hpx::future<ranges::copy_n_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_n_result<FwdIter1, FwdIter2>* otherwise. The *copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter**, typename **F**, typename **P**

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_if_result<typename hpx::traits::range_traits<Rng>
```

Copies the elements in the range, defined by [first, last) to another range beginning at *dest*. Copies only the elements for which the predicate *f* returns true. The order of the elements that are not removed is preserved.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than std::distance(begin(rng), end(rng)) assignments, exactly std::distance(begin(rng), end(rng)) applications of the predicate *f*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for FwdIter1.
- **FwdIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **iter**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type

parallel_policy or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *copy_if* algorithm returns a *hpx::future<ranges::copy_if_result<iterator_t<Rng>, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_if_result<iterator_t<Rng>, FwdIter2>* otherwise. The *copy_if* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng, typename OutIter, typename F, typename Proj = hpx::parallel::util::parallel_if<ExPolicy, hpx::ranges::copy_if_result<typename hpx::traits::range_traits<Rng>, FwdIter2>>>
```

Copies the elements in the range *rng* to another range beginning at *dest*. Copies only the elements for which the predicate *f* returns true. The order of the elements that are not removed is preserved.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than `std::distance(begin(rng), end(rng))` assignments, exactly `std::distance(begin(rng), end(rng))` applications of the predicate *f*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects

passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *copy_if* algorithm returns a *hpx::future<ranges::copy_if_result<iterator_t<Rng>, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_if_result<iterator_t<Rng>, FwdIter2>* otherwise. The *copy_if* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Rng, typename T, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_>
```

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* comparisons.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *T*: The type of the value to search for (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *value*: The value to search for.

- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Note The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *count* algorithm returns a *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIter>::difference_type*). The *count* algorithm returns the number of elements satisfying the given criteria.

```
template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator>
```

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note Complexity: Performs exactly *last - first* applications of the predicate.

Note The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *count_if* algorithm returns *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIter>::difference_type*). The *count* algorithm returns the number of elements satisfying the given criteria.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Rng:** The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F:** The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj:** The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **rng:** Refers to the sequence of elements the algorithm will be applied to.
- **f:** Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

<code>bool pred(const Type &a);</code>

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy>
util::detail::algorithm_result<ExPolicy, typename traits::range_iterator<Rng>::type>::type destroy (ExPolicy &&policy, Rng &&rng)
```

Destroys objects of type `typename iterator_traits<ForwardIt>::value_type` in the range `[first, last)`.

The operations in the parallel `destroy` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` operations.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.

The operations in the parallel `destroy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `destroy` algorithm returns a `hpx::future<void>`, if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `void` otherwise.

```
template<typename ExPolicy, typename FwdIter, typename Size>
util::detail::algorithm_result<ExPolicy, FwdIter>::type destroy_n (ExPolicy &&policy, FwdIter first, Size count)
```

Destroys objects of type `typename iterator_traits<ForwardIt>::value_type` in the range `[first, first + count)`.

The operations in the parallel `destroy_n` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` operations, if `count > 0`, no assignments otherwise.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of elements to apply this algorithm to.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count**: Refers to the number of elements starting at *first* the algorithm will be applied to.

The operations in the parallel *destroy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *destroy_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename Pred, typename Proj1, typename Proj2>
bool ends_with(FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, Pred &&pred, Proj1 &&proj1, Proj2 &&proj2)
```

Checks whether the second range defined by [first1, last1) matches the suffix of the first range defined by [first2, last2)

The assignments in the parallel *ends_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **Iter1**: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1**: The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2**: The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- **Sent2**: The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred**: The binary predicate that compares the projected elements.
- **Proj1**: The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj1**: The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **first1**: Refers to the beginning of the source range.
- **last1**: Sentinel value referring to the end of the source range.

- `first2`: Refers to the beginning of the destination range.
- `last2`: Sentinel value referring to the end of the destination range.
- `pred`: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the two ranges projected by `proj1` and `proj2` respectively.
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate `is` invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate `is` invoked.

Return The `ends_with` algorithm returns `bool`. The `ends_with` algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy    &&policy,
                                                               FwdIter1
                                                               first1, Sent1 last1,
                                                               FwdIter2    first2,
                                                               Sent2 last2, Pred
                                                               &&pred,     Proj1
                                                               &&proj1,    Proj2
                                                               &&proj2)
```

Checks whether the second range defined by `[first1, last1)` matches the suffix of the first range defined by `[first2, last2)`

The assignments in the parallel `ends_with` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Linear: at most `min(N1, N2)` applications of the predicate and both projections.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent1`: The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for `Iter1`.
- `FwdIter2`: The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- `Sent2`: The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for `Iter2`.
- `Pred`: The binary predicate that compares the projected elements.
- `Proj1`: The type of an optional projection function for the source range. This defaults to `util::projection_identity`
- `Proj2`: The type of an optional projection function for the destination range. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the source range.
- `last1`: Sentinel value referring to the end of the source range.
- `first2`: Refers to the beginning of the destination range.
- `last2`: Sentinel value referring to the end of the destination range.
- `pred`: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the two ranges projected by `proj1` and `proj2` respectively.
- `proj1`: Specifies the function (or function object) which will be invoked for each of the ele-

ments in the source range as a projection operation before the actual predicate *is* invoked.

- proj2: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *ends_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

```
template<typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
bool ends_with(Rng1 &&rng1, Rng2 &&rng2, Pred &&pred, Proj1 &&proj1, Proj2 &&proj2)
```

Checks whether the second range *rng2* matches the suffix of the first range *rng1*.

The assignments in the parallel *ends_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- Rng1: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- Rng2: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- Pred: The binary predicate that compares the projected elements.
- Proj1: The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- Proj1: The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- rng1: Refers to the source range.
- rng2: Refers to the destination range.
- pred: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- proj1: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- proj2: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Return The *ends_with* algorithm returns *bool*. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy &&pol-
    icy, Rng1 &&rng1,
    Rng2 &&rng2,
    Pred &&pred,
    Proj1 &&proj1,
    Proj2 &&proj2)
```

Checks whether the second range *rng2* matches the suffix of the first range *rng1*.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type

sequenced_policy execute in sequential order in the calling thread.

Note Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2**: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred**: The binary predicate that compares the projected elements.
- **Proj1**: The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj1**: The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng1**: Refers to the source range.
- **rng2**: Refers to the destination range.
- **pred**: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *ends_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred>
util::detail::algorithm_result<ExPolicy, bool>::type equal(ExPolicy &&policy, Iter1 first1, Sent1
last1, Iter2 first2, Sent2 last2, Pred
&&op = Pred(), Proj1 &&proj1 =
Proj1(), Proj2 &&proj2 = Proj2())
```

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *f*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1**: The type of the source iterators used for the end of the first range (deduced).
- **Iter2**: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2**: The type of the source iterators used for the end of the second range (deduced).
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj1**: The type of an optional projection function applied to the first range. This defaults to `util::projection_identity`
- **Proj2**: The type of an optional projection function applied to the second range. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2**: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2**: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op**: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The two ranges are considered equal if, for every iterator *i* in the range `[first1, last1)`, $*i$ equals $(first2 + (i - first1))$. This overload of *equal* uses operator`==` to determine if two elements are equal.

Return The *equal* algorithm returns a `hpx::future<bool>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `bool` otherwise. The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range `[first1, last1)` does not equal the length of the range `[first2, last2)`, it returns false.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = ranges::equal_to, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity> bool equal(Rng1 &first1, Rng1 &last1, Rng2 &first2, Rng2 &last2, ExPolicy &policy, Pred op = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

```
util::detail::algorithm_result<ExPolicy, bool>::type equal (ExPolicy &&policy, Rng1 &&rng1,
Rng2 &&rng2, Pred &&op = Pred(),
Proj1 &&proj1 = Proj1(), Proj2
&&proj2 = Proj2())
```

Returns true if the range [first1, last1) is equal to the range starting at first2, and false otherwise.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most *last1 - first1* applications of the predicate *f*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng1*: The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Rng2*: The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- *Proj1*: The type of an optional projection function applied to the first range. This defaults to `util::projection_identity`
- *Proj2*: The type of an optional projection function applied to the second range. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng1*: Refers to the first sequence of elements the algorithm will be applied to.
- *rng2*: Refers to the second sequence of elements the algorithm will be applied to.
- *op*: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- *proj2*: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The two ranges are considered equal if, for every iterator *i* in the range [first1, last1), $*i$ equals $(\text{first2} + (i - \text{first1}))$. 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.

```
namespace hpx
```

namespace ranges**Functions**

```
template<typename InIter, typename Sent, typename OutIter, typename T>
exclusive_scan_result<InIter, OutIter> exclusive_scan (InIter first, Sent last, OutIter dest, T
                                                       init)
Assigns through each iterator i in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))
```

The reduce operations in the parallel *exclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *std::plus*<*T*>.

Template Parameters

- **InIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **InIter**.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest**: Refers to the beginning of the destination range.
- **init**: The initial value for the generalized sum.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *exclusive_scan* algorithm returns *util::in_out_result*<*InIter*, *OutIter*>. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, 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 FwdIter1, typename Sent, typename FwdIter2, typename T>
util::detail::algorithm_result<ExPolicy, exclusive_scan_result<FwdIter1, FwdIter2>>::type exclusive_scan (ExPolicy
&&policy,
FwdIter1
first,
Sent
last,
FwdIter2
dest,
T
init)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *dest*: Refers to the beginning of the destination range.
- *init*: The initial value for the generalized sum.

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *exclusive_scan* algorithm returns a `hpx::future<util::in_out_result<FwdIter1, FwdIter2>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `util::in_out_result<FwdIter1, FwdIter2>` otherwise. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, 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 InIter, typename Sent, typename OutIter, typename T, typename Op>
exclusive_scan_result<InIter, OutIter> exclusive_scan(InIter first, Sent last, OutIter dest, T
init, Op &&op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, ..., *(first + (i - result) - 1)).

The reduce operations in the parallel *exclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.

- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).
- `Op`: The type of the binary function object used for the reduction operation.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `dest`: Refers to the beginning of the destination range.
- `init`: The initial value for the generalized sum.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the *i*th input element in the *i*th sum. If `op` is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

Return The `exclusive_scan` algorithm returns `util::in_out_result<InIter, OutIter>`. The `exclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- `a1` when $N = 1$
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \dots, aN))$ where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename T, typename Op
util::detail::algorithm_result<ExPolicy, exclusive_scan_result<FwdIter1, FwdIter2>>::type exclusive_scan(ExPolicy
    &&policy,
    FwdIter1
    first,
    Sent
    last,
    FwdIter2
    dest,
    T
    init,
    Op
    &&op)
```

Assigns through each iterator *i* in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, \dots, *(first + (i - result) - 1)).`

The reduce operations in the parallel `exclusive_scan` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `op`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in

which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter1`.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).
- `Op`: The type of the binary function object used for the reduction operation.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `dest`: Refers to the beginning of the destination range.
- `init`: The initial value for the generalized sum.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel `exclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the *i*th input element in the *i*th sum. If `op` is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

Return The `exclusive_scan` algorithm returns a `hpx::future<util::in_out_result<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `util::in_out_result<FwdIter1, FwdIter2>` otherwise. The `exclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- $a1$ when $N = 1$
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \dots, aN))$ where $1 < K+1 = M \leq N$.

```
template<typename Rng, typename O, typename T>
exclusive_scan_result<traits::range_iterator_t<Rng>, O> exclusive_scan(Rng &&rng, O dest,
T init)
Assigns through each iterator i in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))
```

The reduce operations in the parallel `exclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

Template Parameters

- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **init**: The initial value for the generalized sum.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the i th input element in the i th sum.

Return The *exclusive_scan* algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a₁, ..., a_N) is defined as:

- a₁ when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a₁, ..., a_K)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, a_M, ..., a_N) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename Rng, typename O, typename T>
util::detail::algorithm_result<ExPolicy, exclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type exclusive_scan
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **init**: The initial value for the generalized sum.

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *exclusive_scan* algorithm returns a `hpx::future<util::in_out_result<traits::range_iterator_t<Rng>, O>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `util::in_out_result<traits::range_iterator_t<Rng>, O>` otherwise. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.

```
template<typename Rng, typename O, typename T, typename Op>
exclusive_scan_result<traits::range_iterator_t<Rng>, O> exclusive_scan(Rng &&rng, O dest,
T init, Op &&op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))

The reduce operations in the parallel *exclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *O*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *init*: The initial value for the generalized sum.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *exclusive_scan* algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename Rng, typename O, typename T, typename OpExPolicy, exclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type exclusive_scan
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *std::plus*<*T*>.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).
- **Op**: The type of the binary function object used for the reduction operation.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **init**: The initial value for the generalized sum.
- **op**: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *exclusive_scan* algorithm returns a *hpx::future<util::in_out_result<traits::range_iterator_t<*Rng*>, *O*>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result<traits::range_iterator_t<*Rng*>, *O*>* otherwise. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the

sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.

namespace hpx

Functions

```
template<typename ExPolicy, typename Rng, typename T>
util::detail::algorithm_result<ExPolicy>::type fill(ExPolicy &&policy, Rng &&rng, T const
&value)
```

Assigns the given value to the elements in the range [first, last).

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T**: The type of the value to be assigned (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **value**: The value to be assigned.

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *fill* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*).

```
template<typename ExPolicy, typename Iterator, typename Size, typename T>
util::detail::algorithm_result<ExPolicy, Iterator>::type fill_n(ExPolicy &&policy, Iterator first, Size
count, T const &value)
```

Assigns the given value *value* to the first *count* elements in the range beginning at *first* if *count* > 0. Does nothing otherwise.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, for count > 0.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iterator**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of elements to apply *f* to.
- **T**: The type of the value to be assigned (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count**: Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value**: The value to be assigned.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*).

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Iter, typename Sent, typename T, typename Proj = util::projection_identity,
        util::detail::algorithm_result<ExPolicy, Iter>::type find(ExPolicy &&policy, Iter first, Sent last, T
                                                               const &val, Proj &&proj = Proj())
```

Returns the first element in the range [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most last - first applications of the operator==().

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter**: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for *Iter*.
- **T**: The type of the value to find (deduced).

- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`
- Parameters**

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `val`: the value to compare the elements to
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *find* algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

```
template<typename ExPolicy, typename Rng, typename T, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, Iter>::type find(ExPolicy &&policy, Rng &&rng, T
                                                       const &val, Proj &&proj = Proj())
```

Returns the first element in the range [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most *last - first* applications of the operator `==()`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `T`: The type of the value to find (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `val`: the value to compare the elements to
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *find* algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, Iter1>::type find(ExPolicy &&policy, Iter1 &&first, Sent1 &&last, Iter2 &&val, Proj &&proj = Proj())
```

```
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_end(ExPolicy  
  &&policy,  
  Iter1  
  first1,  
  Sent1  
  last1,  
  Iter2  
  first2,  
  Sent2  
  last2,  
  Pred  
  &&op  
 =  
 Pred(),  
 Proj1  
&&proj1  
 =  
 Proj1(),  
 Proj2  
&&proj2  
 =  
 Proj2())
```

Returns the last subsequence of elements [first2, last2) found in the range [first1, last1) using the given predicate *f* to compare elements.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most $S^*(N-S+1)$ comparisons where $S = \text{distance}(\text{first2}, \text{last2})$ and $N = \text{distance}(\text{first1}, \text{last1})$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1**: The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1**: The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2**: The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2**: The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj1**: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- **Proj2**: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the first sequence of elements the algorithm will be applied to.

- `last1`: Refers to the end of the first sequence of elements the algorithm will be applied to.
- `first2`: Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- `last2`: Refers to the end of the second sequence of elements the algorithm will be applied to.
- `op`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `iterator_t<Rng>` and `iterator_t<Rng2>` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range of type dereferenced `iterator_t<Rng1>` as a projection operation before the function `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range of type dereferenced `iterator_t<Rng2>` as a projection operation before the function `op` is invoked.

The comparison operations in the parallel `find_end` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of `find_end` is available if the user decides to provide the algorithm their own predicate `op`.

Return The `find_end` algorithm returns a `hpx::future<iterator_t<Rng>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `iterator_t<Rng>` otherwise. The `find_end` algorithm returns an iterator to the beginning of the last subsequence `rng2` in range `rng`. If the length of the subsequence `rng2` is greater than the length of the range `rng`, `end(rng)` is returned. Additionally if the size of the subsequence is empty or no subsequence is found, `end(rng)` is also returned.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = ranges::equal_to, typename Proj1
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_end(ExPolicy
&&policy,
Rng1
&&&rng,
Rng2
&&&rng2,
Pred
&&&op
=
Pred(),
Proj1
&&&proj1
=
Proj1(),
Proj2
&&&proj2
=
Proj2())
```

Returns the last subsequence of elements `rng2` found in the range `rng` using the given predicate `f` to compare elements.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most $S^*(N-S+1)$ comparisons where $S = \text{distance}(\text{begin}(\text{rng2}), \text{end}(\text{rng2}))$ 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.
- *Rng1*: The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- *Rng2*: The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- *Proj1*: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- *Proj2*: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the first sequence of elements the algorithm will be applied to.
- *rng2*: Refers to the second sequence of elements the algorithm will be applied to.
- *op*: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of the first range of type dereferenced *iterator_t<Rng1>* as a projection operation before the function *op* is invoked.
- *proj2*: Specifies the function (or function object) which will be invoked for each of the elements of the second range of type dereferenced *iterator_t<Rng2>* as a projection operation before the function *op* is invoked.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *op*.

Return The *find_end* algorithm returns a `hpx::future<iterator_t<Rng>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *iterator_t<Rng>* otherwise. The *find_end* algorithm returns an iterator to the beginning of the last subsequence *rng2* in range *rng*. If the length of the subsequence *rng2* is greater than the length of the range *rng*, *end(rng)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng)* is also returned.

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Proj1, typename Proj2>
```

```
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_first_of (ExPolicy && icy Iter firs Ser las Iter firs Ser las Iter firs Ser las Pre && = Pre Pro && = Pro Pro && = Pro Pro && = Pro
```

Searches the range [first1, last1) for any elements in the range [first2, last2). Uses binary predicate *p* to compare elements

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most ($S \cdot N$) comparisons where $S = \text{distance}(\text{first2}, \text{last2})$ and $N = \text{distance}(\text{first1}, \text{last1})$.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Iter1*: The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent1*: The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for *Iter1*.
- *Iter2*: The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent2*: The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for *Iter2*.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- *Proj1*: The type of an optional projection function. This defaults to *util::projection_identity* and is applied to the elements in *rng1*.
- *Proj2*: The type of an optional projection function. This defaults to *util::projection_identity* and is applied to the elements in *rng2*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the first sequence of elements the algorithm will be applied to.

- `last1`: Refers to the end of the first sequence of elements the algorithm will be applied to.
- `first2`: Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- `last2`: Refers to the end of the second sequence of elements the algorithm will be applied to.
- `op`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `iterator_t<Rng1>` and `iterator_t<Rng2>` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced `iterator_t<Rng1>` before the function `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced `iterator_t<Rng2>` before the function `op` is invoked.

The comparison operations in the parallel `find_first_of` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of `find_first_of` is available if the user decides to provide the algorithm their own predicate `op`.

Return The `find_end` algorithm returns a `hpx::future<iterator_t<Rng1>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `iterator_t<Rng1>` otherwise. The `find_first_of` algorithm returns an iterator to the first element in the range `rng1` that is equal to an element from the range `rng2`. If the length of the subsequence `rng2` is greater than the length of the range `rng1`, `end(rng1)` is returned. Additionally if the size of the subsequence is empty or no subsequence is found, `end(rng1)` is also returned.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = ranges::equal_to, typename Proj1 = util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_first_of(ExPolicy exPolicy, Rng1 rng1, Rng2 rng2, Pred pred, Proj1 proj1) {
    return hpx::make_ready_future<iterator_t<Rng1>>(
        hpx::parallel::for_each(exPolicy, rng1, [rng2 = std::move(rng2), pred = std::move(pred)](auto& elem) {
            return hpx::parallel::find_if(exPolicy, rng2, [&pred](auto& elem2) { return pred(elem, elem2); });
        })
    );
}
```

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.

namespace hpx

namespace ranges

Functions

```
template<typename InIter, typename Sent, typename F, typename Proj = util::projection_identity>
hpx::ranges::for_each_result<InIter, F> for_each(InIter first, Sent last, F &&f, Proj &&proj =
                                                 Proj())
```

Applies *f* to the result of dereferencing every iterator in the range [first, last).

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *last - first* times.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

Return {last, HPX_MOVE(f)} where last is the iterator corresponding to the input sentinel last.

Template Parameters

- *InIter*: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have *const&*. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *last - first* times.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Return {first + count, HPX_MOVE(f)}

Template Parameters

- *InIter*: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `count`: Refers to the number of elements starting at *first* the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have `const&`. The type `Type` must be such that an object of type `InIter` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename F, typename Proj = util::projection_id>  
FwdIter for_each(ExPolicy &&policy, FwdIter first, Sent last, F &&f, Proj &&proj = Proj())
```

Applies *f* to the result of dereferencing every iterator in the range [first, last).

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *last - first* times.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

Return The *for_each* algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. It returns *last*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `FwdIter`: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter`.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of `CopyConstructible`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have `const&`. The type `Type` must be such that an object of type `InIter` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
template<typename Rng, typename F, typename Proj = util::projection_identity>
```

```
hpx::ranges::for_each_result<typename hpx::traits::range_iterator<Rng>::type, F> for_each(ExPolicy
    &&policy,
    Rng
    &&rng,
    F
    &&f,
    Proj
    &&proj
    =
    Proj())
```

Applies f to the result of dereferencing every iterator in the given range rng .

If f returns a result, the result is ignored.

Note Complexity: Applies f exactly $\text{size}(rng)$ times.

If the type of $first$ satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator.

Return {std::end(rng), HPX_MOVE(f)}

Template Parameters

- Rng : The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- F : The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires F to meet the requirements of *CopyConstructible*.
- $Proj$: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- $policy$: The execution policy to use for the scheduling of the iterations.
- rng : Refers to the sequence of elements the algorithm will be applied to.
- f : Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have `const&`. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- $proj$: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type for_each(ExPolicy
    &&policy,
    Rng
    &&rng,
    F
    &&f,
    Proj
    &&proj
    =
    Proj())
```

Applies f to the result of dereferencing every iterator in the given range rng .

If f returns a result, the result is ignored.

Note Complexity: Applies f exactly $\text{size}(\text{rng})$ times.

If the type of first satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have **const&**. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *for_each* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename F, typename Proj = util::projection_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.

Return The *for_each* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *FwdIter*: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have *const&*. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename Iter, typename Sent, typename ...Args>
void for_loop(Iter first, Sent last, Args&&... args)
```

The *for_loop* implements loop functionality over a range specified by iterator bounds. These algorithms resemble *for_each* from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of *for_loop* without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *Iter* shall meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Template Parameters

- *Iter*: The type of the iteration variable (input iterator).
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *Iter*.
- *Args*: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.

- **args**: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Iter const& a, ...);
```

The signature does not need to have **const**&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

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 Iter, typename Sent, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type for_loop (ExPolicy &&policy, Iter first, Sent last,
Args&&... args)
```

The **for_loop** implements loop functionality over a range specified by iterator bounds. These algorithms resemble **for_each** from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: *Iter* shall meet the requirements of a forward iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter**: The type of the iteration variable (forward iterator).
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *Iter*.
- **Args**: A parameter pack, 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(Iter const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies `f` to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the `args` parameter pack. The length of the input sequence is `last - first`.

The first element in the input sequence is specified by `first`. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the `args` parameter pack excluding `f`, an additional argument is passed to each application of `f` as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of `f`, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of `f` in the input sequence.

Complexity: Applies `f` exactly once for each element of the input sequence.

Remarks: If `f` returns a result, the result is ignored.

Return The `for_loop` algorithm returns a `hpx::future<void>` if the execution policy is of type `hpx::execution::sequenced_task_policy` or `hpx::execution::parallel_task_policy` and returns `void` otherwise.

```
template<typename Rng, typename ...Args>
void for_loop(Rng &&rng, Args&&... args)
```

The `for_loop` implements loop functionality over a range specified by a range. These algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of `for_loop` without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Requires: `Rng::iterator` shall meet the requirements of an input iterator type. The `args` parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, `f`. `f` shall meet the requirements of `MoveConstructible`.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

- **Args**: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **rng**: Refers to theof the sequence of elements the algorithm will be applied to.
- **args**: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last] should expose a signature equivalent to:

```
<ignored> pred(Rng::iterator const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

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 Rng, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type for_loop(ExPolicy &&policy, Rng &&rng,
                                                       Args&&... args)
```

The `for_loop` implements loop functionality over a range specified by a range. These algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: `Rng::iterator` shall meet the requirements of a forward iterator type. The $args$ parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, f . f shall meet the requirements of `MoveConstructible`.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

- **Args:** A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **rng:** Refers to theof the sequence of elements the algorithm will be applied to.
- **args:** The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last] should expose a signature equivalent to:

```
<ignored> pred(Rng::iterator const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Return The *for_loop* algorithm returns a `hpx::future<void>` if the execution policy is of type `hpx::execution::sequenced_task_policy` or `hpx::execution::parallel_task_policy` and returns `void` otherwise.

```
template<typename Iter, typename Sent, typename S, typename ...Args>
void for_loop_strided(Iter first, Sent last, S stride, Args&&... args)
```

The *for_loop_strided* implements loop functionality over a range specified by iterator bounds. These algorithms resemble *for_each* from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of *for_loop_strided* without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Requires: *Iter* shall meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Template Parameters

- *Iter*: The type of the iteration variable (input iterator).
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *Iter*.
- *S*: The type of the stride variable. This should be an integral type.
- *Args*: A parameter pack, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *stride*: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if *Iter* meets the requirements a bidirectional iterator.
- *args*: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last] should expose a signature equivalent to:

```
<ignored> pred(Iter const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last - first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

```
template<typename ExPolicy, typename Iter, typename Sent, typename S, typename... A>
```

The `for_loop_strided` implements loop functionality over a range specified by iterator bounds. These algorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: *Iter* shall meet the requirements of a forward iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of `MoveConstructible`.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter**: The type of the iteration variable (forward iterator).
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for Iter.
- **S**: The type of the stride variable. This should be an integral type.
- **Args**: A parameter pack, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **stride**: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if Iter meets the requirements a bidirectional iterator.
- **args**: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Iter const& a, ...);
```

The signature does not need to have **const**&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Return The *for_loop_strided* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

```
template<typename Rng, typename S, typename ...Args>
void for_loop_strided(Rng &&rng, S stride, Args&&... args)
```

The *for_loop_strided* implements loop functionality over a range specified by a range. These algorithms resemble *for_each* from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of `for_loop_strided` without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Requires: `Rng::iterator` shall meet the requirements of an input iterator type. The `args` parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, `f`. `f` shall meet the requirements of `MoveConstructible`.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `S`: The type of the stride variable. This should be an integral type.
- `Args`: A parameter pack, 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

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `stride`: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if `Rng::iterator` meets the requirements 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(Rng::iterator const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies `f` to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the `args` parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by `first`. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the `args` parameter pack excluding `f`, an additional argument is passed to each application of `f` as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of `f`, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of `f` in the input sequence.

Complexity: Applies `f` exactly once for each element of the input sequence.

Remarks: If `f` returns a result, the result is ignored.

```
template<typename ExPolicy, typename Rng, typename S, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type for_loop_strided(ExPolicy &&policy, Rng
&&rng, S stride, Args&&...
args)
```

The `for_loop_strided` implements loop functionality over a range specified by a range. These al-

gorithms resemble `for_each` from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: `Rng::iterator` shall meet the requirements of a forward iterator type. The `args` parameter pack shall have at least one element, comprising objects returned by invocations of `reduction` and/or `induction` function templates followed by exactly one element invocable element-access function, `f`. `f` shall meet the requirements of `MoveConstructible`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `S`: The type of the stride variable. This should be an integral type.
- `Args`: A parameter pack, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `stride`: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if `Rng::iterator` meets the requirements 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(Rng::iterator const& a, ...);
```

The signature does not need to have `const&`. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Effects: Applies `f` to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the `args` parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by `first`. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the `args` parameter pack excluding `f`, an additional argument is passed to each application of `f` as follows:

Note As described in the C++ standard, arithmetic on non-random-access iterators is performed using `advance` and `distance`.

Note The order of the elements of the input sequence is important for determining ordinal position of an application of `f`, even though the applications themselves may be unordered.

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of `f` in the input sequence.

Complexity: Applies `f` exactly once for each element of the input sequence.

Remarks: If `f` returns a result, the result is ignored.

Return The *for_loop_strided* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename Rng, typename F>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type generate(ExPolicy &&policy,
Rng &&rng,
F &&f)
```

Assign each element in range [first, last) a value generated by the given function object f

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *f*: generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

```
template<typename ExPolicy, typename Iter, typename Sent, typename F>
util::detail::algorithm_result<ExPolicy, Iter>::type generate(ExPolicy &&policy, Iter first, Sent last, F &&f)
```

Assign each element in range [first, last) a value generated by the given function object f

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly $distance(first, last)$ invocations of f and assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter**: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires F to meet the requirements of *CopyConstructible*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **f**: generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type generate_n(ExPolicy &&policy, FwdIter
first, Size count, F &&f)
```

Assigns each element in range [first, first+count) a value generated by the given function object *g*.

The assignments in the parallel *generate_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly $count$ invocations of f and assignments, for $count > 0$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires F to meet the requirements of *CopyConstructible*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count**: Refers to the number of elements in the sequence the algorithm will be applied to.

- `f`: Refers to the generator function object that will be called. The signature of the function should be equivalent to

```
Ret fun();
```

The type `Ret` must be such that an object of type `OutputIt` can be dereferenced and assigned a value of type `Ret`.

The assignments in the parallel `generate_n` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `replace_if` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. It returns `last`.

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred>
util::detail::algorithm_result<ExPolicy, bool>::type includes(ExPolicy &&policy, Iter1
first1, Sent1 last1, Iter2 first2,
Sent2 last2, Pred &&op =
Pred(), Proj1 &&proj1 =
Proj1(), Proj2 &&proj2 =
Proj2())
```

Returns true if every element from the sorted range `[first2, last2]` is found within the sorted range `[first1, last1]`. Also returns true if `[first2, last2]` is empty. The version expects both ranges to be sorted with the user supplied binary predicate `f`.

The comparison operations in the parallel `includes` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note At most $2^*(N1+N2-1)$ comparisons, where $N1 = \text{std}::\text{distance}(first1, last1)$ and $N2 = \text{std}::\text{distance}(first2, last2)$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Iter1`: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- `Sent1`: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for `Iter1`.
- `Iter2`: The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- `Sent2`: The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for `Iter2`.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `includes` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`
- `Proj1`: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`

- `Proj2`: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `op`: The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate `op` is invoked.

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 Rng1, typename Rng2, typename Pred = detail::less, typename Proj1 = util::detail::algorithm_result<ExPolicy, bool>::type::type includes(ExPolicy &&policy, Rng1 &&rng1, Rng2 &&rng2, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Returns true if every element from the sorted range $[first2, last2)$ is found within the sorted range $[first1, last1)$. Also returns true if $[first2, last2)$ is empty. The version expects both ranges to be sorted with the user supplied binary predicate `f`.

The comparison operations in the parallel `includes` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note At most $2^*(N1+N2-1)$ comparisons, where $N1 = \text{std::distance}(first1, last1)$ and $N2 = \text{std::distance}(first2, last2)$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1`: The type of the source range used (deduced). The iterators extracted from this range

type must meet the requirements of an input iterator.

- `Rng2`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `includes` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`
- `Proj1`: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- `Proj2`: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the first sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the second sequence of elements the algorithm will be applied to.
- `op`: The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate `op` is invoked.

The comparison operations in the parallel `includes` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `includes` algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `bool` otherwise. The `includes` algorithm returns true every element from the sorted range `[first2, last2)` is found within the sorted range `[first1, last1)`. Also returns true if `[first2, last2)` is empty.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename InIter, typename Sent, typename OutIter>
inclusive_scan_result<InIter, OutIter> inclusive_scan(InIter first, Sent last, OutIter dest)
Assigns through each iterator i in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, *first, ..., *(first + (i - result))).
```

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `op`.

Template Parameters

- **InIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **InIter**.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest**: Refers to the beginning of the destination range.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Return The *inclusive_scan* algorithm returns *util::in_out_result<InIter, OutIter>*. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2>
util::detail::algorithm_result<ExPolicy, inclusive_scan_result<FwdIter1, FwdIter2>>::type inclusive_scan(ExPolicy
&&policy,
FwdIter1
first,
Sent
last,
FwdIter2
dest)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, *first, ..., *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **FwdIter**.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

- `dest`: Refers to the beginning of the destination range.

The reduce operations in the parallel `inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i th input element in the i th sum.

Return The `inclusive_scan` algorithm returns a `hpx::future<util::in_out_result<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `util::in_out_result<FwdIter1, FwdIter2>` otherwise. The `inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN)` is defined as:

- $a1$ when N is 1
- `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)`
 - `GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN)` where $1 < K+1 = M \leq N$.

```
template<typename Rng, typename O>
inclusive_scan_result<traits::range_iterator_t<Rng>, O> inclusive_scan(Rng&&rng, O
dest)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(+, *first, ..., *(first + (i - result)))`.

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `op`.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `O`: The type of the iterator representing the destination range (deduced).

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i th input element in the i th sum.

Return The `inclusive_scan` algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`. The `inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(+, 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 Rng, typename O>
util::detail::algorithm_result<ExPolicy, inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type inclusive_scan
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM($+, *first, \dots, *(first + (i - result))$).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate op .

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *O*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

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 reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the i th input element in the i th sum.

Return The *inclusive_scan* algorithm returns a *hpx::future<util::in_out_result <traits::range_iterator_t<Rng>, O>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result <traits::range_iterator_t<Rng>, O>* otherwise. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM($+, a_1, \dots, a_N$) is defined as:

- a_1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM($+, a_1, \dots, a_K$)
 - GENERALIZED_NONCOMMUTATIVE_SUM($+, a_M, \dots, a_N$) where $1 < K+1 = M \leq N$.

```
template<typename InIter, typename Sent, typename OutIter, typename Op>
inclusive_scan_result<InIter, OutIter> inclusive_scan(InIter first, Sent last, OutIter dest, Op
&&op)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM($op, *first, \dots, *(first + (i - result))$).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate op .

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

- `Op`: The type of the binary function object used for the reduction operation.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `dest`: Refers to the beginning of the destination range.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the *i*th input element in the *i*th sum.

Return The `inclusive_scan` algorithm returns `util::in_out_result<InIter, OutIter>`. The `inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename OpExPolicy, inclusive_scan_result<FwdIter1, FwdIter2>>::type inclusive_scan(ExPolicy
    &&policy,
    FwdIter1
    first,
    Sent
    last,
    FwdIter2
    dest,
    Op
    &&op)
```

Assigns through each iterator *i* in [`result`, `result` + (`last` - `first`)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(`op`, *`first`, ..., *(`first` + (*i* - `result`))).

The reduce operations in the parallel `inclusive_scan` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `op`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter`.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator

type must meet the requirements of an forward iterator.

- **Op:** The type of the binary function object used for the reduction operation.

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **first:** Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last:** Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest:** Refers to the beginning of the destination range.
- **op:** Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun (const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel `inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the *i*th input element in the *i*th sum.

Return The `inclusive_scan` algorithm returns a `hpx::future<util::in_out_result<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `util::in_out_result<FwdIter1, FwdIter2>` otherwise. The `inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN)` is defined as:

- a_1 when N is 1
- `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK)`
 - `GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN)` where $1 < K+1 = M \leq N$.

```
template<typename Rng, typename O, typename Op>
inclusive_scan_result<traits::range_iterator_t<Rng>, O> inclusive_scan (Rng &&rng, O dest,
                                                                     Op &&op)
```

Assigns through each iterator *i* in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, ..., *(first + (i - result)))`.

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `op`.

Template Parameters

- `Rng:` The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `O:` The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Op:` The type of the binary function object used for the reduction operation.

Parameters

- `rng:` Refers to the sequence of elements the algorithm will be applied to.
- `dest:` Refers to the beginning of the destination range.
- `op:` Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be

equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the *i*th input element in the *i*th sum.

Return The `inclusive_scan` algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`. The `inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(+, 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 Rng, typename O, typename Op>
util::detail::algorithm_result<ExPolicy, inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type inclusive_scan
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, ..., *(first + (i - result))).`

The reduce operations in the parallel `inclusive_scan` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate `op`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `O`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Op`: The type of the binary function object used for the reduction operation.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
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 `inclusive_scan` algorithm returns a `hpx::future<util::in_out_result<traits::range_iterator_t<Rng>, O>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `util::in_out_result<traits::range_iterator_t<Rng>, O>` otherwise. The `inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN)` is defined as:

- $a1$ when N is 1
- `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK)`
 - `GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN)` where $1 < K+1 = M \leq N$.

```
template<typename InIter, typename Sent, typename OutIter, typename T, typename Op>
inclusive_scan_result<InIter, OutIter> inclusive_scan(InIter first, Sent last, OutIter dest, Op
&&op, T init)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, ..., *(first + (i - result)))`.

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter`.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Op`: The type of the binary function object used for the reduction operation.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `dest`: Refers to the beginning of the destination range.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- `init`: The initial value for the generalized sum.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Return The *inclusive_scan* algorithm returns *util::in_out_result<InIter, OutIter>*. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ when *N* is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*_M, ..., *a*_N)) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename T, typename Op>
util::detail::algorithm_result<ExPolicy, inclusive_scan_result<FwdIter1, FwdIter2>>::type inclusive_scan(ExPolicy
    &&policy,
    FwdIter1
    first,
    Sent
    last,
    FwdIter2
    dest,
    T
    init,
    Op
    &&op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *init*, **first*, ..., *(*first* + (*i* - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).
- **Op**: The type of the binary function object used for the reduction operation.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest**: Refers to the beginning of the destination range.
- **init**: The initial value for the generalized sum.
- **op**: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be

equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel `inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i th input element in the i th sum. If op is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

Return The `inclusive_scan` algorithm returns a `hpx::future<util::in_out_result<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `util::in_out_result<FwdIter1, FwdIter2>` otherwise. The `inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_N) is defined as:

- a_1 when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_{M+1}, \dots, a_N))$ where $1 < K+1 = M \leq N$.

```
template<typename Rng, typename O, typename Op, typename T>
inclusive_scan_result<traits::range_iterator_t<Rng>, O> inclusive_scan (Rng &&rng, O dest,
                                                                     Op &&op, T init)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, \dots, *(first + (i - result)))`.

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate op .

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `O`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Op`: The type of the binary function object used for the reduction operation.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- `init`: The initial value for the generalized sum.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Return The *inclusive_scan* algorithm returns *util::in_out_result<traits::range_iterator_t<Rng>, O>*

The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ when *N* is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*_M, ..., *a*_N)) where $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename Rng, typename O, typename Op, typename T>
```

```
util::detail::algorithm_result<ExPolicy, inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type inclusive_sc
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(*op*, init, *first, ..., *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).
- **Op**: The type of the binary function object used for the reduction operation.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **init**: The initial value for the generalized sum.
- **op**: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input

sequence can be implicitly converted to any of those types.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Return The *inclusive_scan* algorithm returns a *hpx::future<util::in_out_result<traits::range_iterator_t<Rng>, O>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result<traits::range_iterator_t<Rng>, O>* otherwise. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*1, ..., *a*N) is defined as:

- *a*1 when N is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*1, ..., *a*K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*M, ..., *a*N)) where 1 < K+1 = M <= N.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename Iter, typename Sent, typename Comp = detail::less, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type is_heap(ExPolicy &&policy, Iter first, Sent last,
                                                               Comp &&comp = Comp(), Proj &&proj
                                                               = Proj())
```

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter**: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **iter**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent**: Refers to the end of the sequence of elements the algorithm will be applied to.

- `comp`: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *is_heap* algorithm returns a `hpx::future<bool>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `bool` otherwise. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

```
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 an random access iterator.
- `Comp`: The type of the function/function object to use (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `comp`: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *is_heap* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

```
template<typename ExPolicy, typename Iter, typename Sent, typename Comp = detail::less, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, Iter>::type is_heap_until(ExPolicy &&policy, Iter first,
                                                               Sent sent, Comp &&comp =
                                                               Comp(), Proj &&proj = Proj())
```

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [*first*, *it*) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = *last - first*.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter:** The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent:** The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp:** The type of the function/function object to use (deduced).
- **Proj:** The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **iter:** Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent:** Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp:** *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *is_heap_until* algorithm returns a *hpx::future<RandIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandIter* otherwise. The *is_heap_until* algorithm returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [*first*, *it*) is a max heap.

```
template<typename ExPolicy, typename Rng, typename Comp = detail::less, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type is_heap_until(ExPolicy  
    &&policy,  
    Rng  
    &&rng,  
    Comp  
    &&comp  
    =  
    Comp(),  
    Proj  
    &&proj  
    =  
    Proj())
```

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [*first*, *it*) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- *Comp*: The type of the function/function object to use (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *comp*: *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* 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.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter, typename Sent, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
bool is_sorted(FwdIter first, Sent last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Determines if the range [first, last) is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Pred*: The type of an optional function/function object to use.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- *pred*: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *is_sorted* algorithm returns a *bool*. The *is_sorted* algorithm returns true if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type is_sorted(ExPolicy &&policy, FwdIter first, Sent last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Determines if the range [first, last) is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<T>`
- *Proj*: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- *pred*: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *is_sorted* algorithm returns a `hpx::future<bool>` if the execution policy is of type *task_execution_policy* and returns `bool` otherwise. The *is_sorted* algorithm returns a `bool` if each element in the sequence $[\text{first}, \text{last}]$ satisfies the predicate passed. If the range $[\text{first}, \text{last}]$ contains less than two elements, the function always returns true.

```
template<typename Rng, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
bool is_sorted(ExPolicy &&policy, Rng &&rng, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Determines if the range *rng* is sorted. Uses *pred* to compare elements.

The comparison operations in the parallel *is_sorted* algorithm executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{size}(\text{rng})$. $S = \text{number of partitions}$

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Pred*: The type of an optional function/function object to use.
- *Proj*: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.

- **pred:** Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type` must be such that objects of types `FwdIter` can be dereferenced and then implicitly converted to `Type`.

- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The `is_sorted` algorithm returns a `bool`. The `is_sorted` algorithm returns true if each element in the `rng` satisfies the predicate passed. If the range `rng` contains less than two elements, the function always returns true.

```
template<typename ExPolicy, typename Rng, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type is_sorted(ExPolicy &&pol-  
ic平, Rng &&rng,  
Pred &&pred  
= Pred(), Proj  
&&proj = Proj())
```

Determines if the range `rng` is sorted. Uses `pred` to compare elements.

The comparison operations in the parallel `is_sorted` algorithm invoked with an execution policy object of type `sequenced_policy` executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{size}(\text{rng})$. $S = \text{number of partitions}$

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng:** The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred:** The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `is_sorted` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`
- **Proj:** The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **rng:** Refers to the sequence of elements the algorithm will be applied to.
- **pred:** Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type` must be such that objects of types `FwdIter` can be dereferenced and then implicitly converted to `Type`.

- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel `is_sorted` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *is_sorted* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_sorted* algorithm returns a *bool* if each element in the range *rng* satisfies the predicate passed. If the range *rng* contains less than two elements, the function always returns true.

```
template<typename FwdIter, typename Sent, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::v1::detail::projection_identity>  
FwdIter is_sorted_until(FwdIter first, Sent last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Returns the first element in the range [*first*, *last*) that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm execute in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Pred*: The type of an optional function/function object to use.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- *pred*: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *is_sorted_until* algorithm returns a *FwdIter*. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, *last* is returned.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::v1::detail::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type is_sorted_until(ExPolicy
    &&pol-
    icy,
    FwdIter
    first,
    Sent
    last,
    Pred
    &&pred
    =
    Pred(),
    Proj
    &&proj
    =
    Proj())
```

Returns the first element in the range [first, last) that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted_until* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred**: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

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 Rng, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection<hpx::traits::range_iterator<Rng>::type>::type is_sorted_until(ExPolicy &&policy, Rng &&rng,
Pred &&pred = Pred(), Proj &&proj
= Proj())
```

Returns the first element in the range rng that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm execute in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{size}(\text{rng})$. $S = \text{number of partitions}$

Template Parameters

- **Rng:** The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred:** The type of an optional function/function object to use.
- **Proj:** The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **rng:** Refers to the sequence of elements the algorithm will be applied to.
- **pred:** Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *is_sorted_until* algorithm returns a *FwdIter*. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, last is returned.

```
template<typename ExPolicy, typename Rng, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type is_sorted_until(ExPolicy &&policy, Rng &&rng,
Pred &&pred = Pred(), Proj &&proj
= Proj())
```

Returns the first element in the range rng that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy

object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: at most $(N+S-1)$ comparisons where $N = \text{size}(\text{rng})$. $S = \text{number of partitions}$

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted_until* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **pred**: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *is_sorted_until* algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, *last* is returned.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename InIter1, typename Sent1, typename InIter2, typename Sent2, typename Proj1 = hpx::parallel::task_execution_policy>
bool lexicographical_compare(InIter1 first1, Sent1 last1, InIter2 first2, Sent2 last2, Pred
    &&pred = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2
    = Proj2())
```

Checks if the first range $[first1, last1]$ is lexicographically less than the second range $[first2, last2]$. uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std::distance}(first1, last1)$ and $N2 = \text{std::distance}(first2, last2)$.

Template Parameters

- `InIter1`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input iterator.
- `Sent1`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter1`.
- `InIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an input iterator.
- `Sent2`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter2`.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `lexicographical_compare` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`
- `Proj1`: The type of an optional projection function for `FwdIter1`. This defaults to `util::projection_identity`
- `Proj2`: The type of an optional projection function for `FwdIter2`. This defaults to `util::projection_identity`

Parameters

- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `pred`: Refers to the comparison function that the first and second ranges will be applied to
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel `lexicographical_compare` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Return The `lexicographically_compare` algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `bool` otherwise. The `lexicographically_compare` algorithm returns true if the first range is lexicographically less, otherwise it returns false. range `[first2, last2)`, it returns false.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**, typename **Proj1**, typename **Proj2**, typename **Pred**> bool hpx::parallel::lexicographical_compare(**ExPolicy** exec, **FwdIter1** first1, **Sent1** last1, **FwdIter2** first2, **Sent2** last2, **Proj1** proj1 = util::projection_identity, **Proj2** proj2 = util::projection_identity, **Pred** pred = std::less<>);

```
parallel::util::detail::algorithm_result<ExPolicy, bool>::type lexicographical_compare (ExPolicy
&&pol-
icy,
FwdIter1
first1,
Sent1
last1,
FwdIter2
first2,
Sent2
last2,
Pred
&&pred
=
Pred(),
Proj1
&&proj1
=
Proj1(),
Proj2
&&proj2
=
Proj2())

```

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std}::\text{distance}(\text{first1}, \text{last1})$ and $N2 = \text{std}::\text{distance}(\text{first2}, \text{last2})$.

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.
- **Sent1**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter1.
- **FwdIter2**: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter2.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- **Proj1**: The type of an optional projection function for FwdIter1. This defaults to `util::projection_identity`
- **Proj2**: The type of an optional projection function for FwdIter2. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.

- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `pred`: Refers to the comparison function that the first and second ranges will be applied to
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Return The *lexicographically_compare* algorithm returns a `hpx::future<bool>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `bool` otherwise. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

```
template<typename Rng1, typename Rng2, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
bool lexicographical_compare(Rng1 &&rng1, Rng2 &&rng2, Pred &&pred = Pred(),
                           Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2());
```

Checks if the first range `rng1` is lexicographically less than the second range `rng2`. uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std}::distance(\text{std}::begin(rng1), \text{std}::end(rng1))$ and $N2 = \text{std}::distance(\text{std}::begin(rng2), \text{std}::end(rng2))$.

Template Parameters

- `Rng1`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Rng2`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires `Pred` to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- `Proj1`: The type of an optional projection function for elements of the first range. This defaults to *util::projection_identity*
- `Proj2`: The type of an optional projection function for elements of the second range. This defaults to *util::projection_identity*

Parameters

- `rng1`: Refers to the sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the sequence of elements the algorithm will be applied to.
- `pred`: Refers to the comparison function that the first and second ranges will be applied to
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Note Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Return The *lexicographically_compare* algorithm returns *bool*. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Proj1 = hpx::parallel::util::projection_idem<ExPolicy, Rng1, Rng2>, typename Proj2 = hpx::parallel::util::projection_idem<ExPolicy, Rng2, Rng1>, typename Pred = hpx::parallel::util::projection_idem<ExPolicy, Pred, Pred>, typename Proj1 = hpx::parallel::util::projection_idem<ExPolicy, Proj1, Proj1>, typename Proj2 = hpx::parallel::util::projection_idem<ExPolicy, Proj2, Proj2>>
    bool lexicographical_compare(ExPolicy &&policy, Rng1 &&rng1, Rng2 &&rng2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2());
```

Checks if the first range `rng1` is lexicographically less than the second range `rng2`. uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std}::distance(\text{std}::begin(rng1), \text{std}::end(rng1))$ and $N2 = \text{std}::distance(\text{std}::begin(rng2), \text{std}::end(rng2))$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- Rng1: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- Rng2: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- Pred: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- Proj1: The type of an optional projection function for elements of the first range. This defaults to *util::projection_identity*
- Proj2: The type of an optional projection function for elements of the second range. This defaults to *util::projection_identity*

Parameters

- policy: The execution policy to use for the scheduling of the iterations.
- rng1: Refers to the sequence of elements the algorithm will be applied to.
- rng2: Refers to the sequence of elements the algorithm will be applied to.
- pred: Refers to the comparison function that the first and second ranges will be applied to
- proj1: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- proj2: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Return The *lexicographically_compare* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Rng, typename Comp, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type make_heap (ExPolicy
&&policy,
Rng
&&rng,
Comp
&&comp,
Proj
&&proj =
Proj{ })
```

Constructs a *max heap* in the range [first, last).

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

Note Complexity: at most $(3*N)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$.

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.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **comp**: Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

```
bool comp (const Type &a, const Type &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *RndIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *make_heap* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. It returns *last*.

```
template<typename ExPolicy, typename Rng, typename Proj = util::projection_identity>
```

```
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type make_heap (ExPolicy  
    &&policy,  
    Rng  
    &&rng,  
    Proj  
    &&proj  
    =  
    Proj{})
```

Constructs a *max heap* in the range [first, last). Uses the operator < for comparisons.

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

Note Complexity: at most (3*N) comparisons where $N = \text{distance}(\text{first}, \text{last})$.

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.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *proj*: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *make_heap* algorithm returns a *hpx::future<void>* if the execution policy is of type *task_execution_policy* and returns *void* otherwise.

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent, typename Iter2, typename Sent2, typename Iter3,
```

```
util::detail::algorithm_result<ExPolicy, hpx::ranges::merge_result<Iter1, Iter2, Iter3>>::type merge (ExPolicy
    &&policy,
    Iter1
    first1,
    Sent1
    last1,
    Iter2
    first2,
    Sent2
    last2,
    Iter3
    dest,
    Comp
    &&comp
    =
    Comp(),
    Proj1
    &&proj1
    =
    Proj1(),
    Proj2
    &&proj2
    =
    Proj2())

```

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs $O(\text{std}::\text{distance}(\text{first1}, \text{last1}) + \text{std}::\text{distance}(\text{first2}, \text{last2}))$ 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.
- *Iter1*: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an random access iterator.
- *Sent1*: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for *Iter1*.
- *Iter2*: The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an random access iterator.
- *Sent2*: The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for *Iter2*.
- *Iter3*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- *Comp*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- *Proj1*: The type of an optional projection function to be used for elements of the first range. This defaults to `util::projection_identity`

- `Proj2`: The type of an optional projection function to be used for elements of the second range. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `comp`: `comp` is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `RandIter1` and `RandIter2` can be dereferenced and then implicitly converted to both `Type1` and `Type2`

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual comparison `comp` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual comparison `comp` is invoked.

The assignments in the parallel `merge` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `merge` algorithm returns a `hpx::future<merge_result<Iter1, Iter2, Iter3>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `merge_result<Iter1, Iter2, Iter3>` otherwise. The `merge` algorithm returns the tuple of the source iterator `last1`, the source iterator `last2`, the destination iterator to the end of the `dest` range.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename RandIter3, typename Comp = hpx::range
```

```
util::detail::algorithm_result<ExPolicy, hpx::ranges::merge_result<typename hpx::traits::range_iterator<Rng1>::type, ty
```

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 $O(\text{std::distance(first1, last1)} + \text{std::distance(first2, last2)})$ applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1**: The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Rng2**: The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **RandIter3**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- **Proj1**: The type of an optional projection function to be used for elements of the first range. This defaults to `util::projection_identity`
- **Proj2**: The type of an optional projection function to be used for elements of the second range. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng1**: Refers to the first range of elements the algorithm will be applied to.
- **rng2**: Refers to the second range of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.

- `comp`: *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter1* and *RandIter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*.

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual comparison *comp* is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual comparison *comp* is invoked.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *merge* algorithm returns a *hpx::future<merge_result<RandIter1, RandIter2, RandIter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *merge_result<RandIter1, RandIter2, RandIter3>* otherwise. The *merge* algorithm returns the tuple of the source iterator *last1*, the source iterator *last2*, the destination iterator to the end of the *dest* range.

```
template<typename ExPolicy, typename Iter, typename Sent, typename Comp = hpx::ranges::less, typename Proj = util::detail::algorithm_result<ExPolicy, Iter>::type> inline merge(ExPolicy &&policy, Iter first, Iter middle, Sent last, Comp &&comp = Comp(), Proj &&proj = Proj())
```

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs $O(\text{std}::\text{distance}(\text{first}, \text{last}))$ 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.
- `Iter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- `Sent`: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for `Iter1`.
- `Comp`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires `Comp` to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the first sorted range the algorithm will be applied to.
- `middle`: Refers to the end of the first sorted range and the beginning of the second sorted

range the algorithm will be applied to.

- *last*: Refers to the end of the second sorted range the algorithm will be applied to.
- *comp*: *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *inplace_merge* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. The *inplace_merge* algorithm returns the source iterator *last*

```
template<typename ExPolicy, typename Rng, typename RandIter, typename Comp = hpx::ranges::less, typename Proj = util::detail::algorithm_result<ExPolicy, RandIter>::type inplace_merge(ExPolicy &&policy,  
                                Rng &&rng, RandIter middle, Comp  
                                &&comp = Comp(),  
                                Proj &&proj = Proj())
```

Merges two consecutive sorted ranges [*first*, *middle*) and [*middle*, *last*) into one sorted range [*first*, *last*). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs O(std::distance(*first*, *lastcomp* 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 type `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`

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Sent, typename F, typename Proj = hpx::parallel::util::projection_identity>
FwdIter min_element(FwdIter first, Sent last, F &&f = F(), Proj &&proj = Proj())
```

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel `min_element` algorithm execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$.

Template Parameters

- `FwdIter`: The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `F`: The type of the function/function object to use (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: 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.

Return The *min_element* algorithm returns *FwdIter*. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

```
template<typename Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator_t<Rng> min_element (Rng &&rng, F &&f = F(), Proj &&proj = Proj())
    Finds the smallest element in the range [first, last) using the given comparison function f.
```

The comparisons in the parallel *min_element* algorithm execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std}:\text{distance}(\text{first}, \text{last})$.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *F*: The type of the function/function object to use (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *f*: The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *min_element* algorithm returns a *hpx::traits::range_iterator<Rng>::type* otherwise. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename F, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type min_element (ExPolicy &&policy,
    FwdIter first, Sent sent, 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 $N = \text{std}:\text{distance}(\text{first}, \text{last})$.

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 iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for **FwdIter**.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *min_element* requires **F** to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *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 Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng>>::type min_element(ExPolicy
    &&pol-
    icy,
    Rng
    &&rng,
    F
    &&f
    =
    F(),
    Proj
    &&proj
    =
    Proj())
```

Finds the smallest element in the range [first, last) using the given comparison function f .

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std}:\text{distance}(\text{first}, \text{last})$.

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.
- **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<hpx::traits::range_iterator<Rng>::type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

```
template<typename FwdIter, typename Sent, typename F, typename Proj = hpx::parallel::util::projection_identity>
FwdIter max_element(FwdIter first, Sent sent, F &&f = F(), Proj &&proj = Proj())
```

Finds the greatest element in the range [first, last) using the given comparison function f .

The comparisons in the parallel *max_element* algorithm execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std}:\text{distance}(\text{first}, \text{last})$.

Template Parameters

- `FwdIter`: The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `F`: The type of the function/function object to use (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: 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.

Return The `max_element` algorithm returns a `FwdIter`. The `max_element` algorithm returns the iterator to the smallest element in the range `[first, last)`. If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns `last` if the range is empty.

```
template<typename Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator_t<Rng> max_element (Rng &&rng, F &&f = F(), Proj &&proj = Proj())
    Finds the greatest element in the range [first, last) using the given comparison function f.
```

The comparisons in the parallel `max_element` algorithm execute in sequential order in the calling thread.

Note Complexity: Exactly $\max(N-1, 0)$ comparisons, where $N = \text{std::distance(first, last)}$.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `F`: The type of the function/function object to use (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `f`: The binary predicate which returns true if the This argument is optional and defaults to `std::less`. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type1` must be such that objects of type `FwdIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `f` is invoked.

Return The `max_element` algorithm returns a `hpx::traits::range_iterator<Rng>::type` otherwise. The `max_element` algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename F, typename Proj = hpx::parallel::util::projection_identity<ExPolicy>, typename ProjF = hpx::parallel::util::projection<ExPolicy, F>, typename ProjProjF = hpx::parallel::util::projection<ProjF, Proj>>
Sent max_element(ExPolicy policy, FwdIter first, Sent sent, F f, 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 $N = \text{std}::distance(\text{first}, \text{last})$.

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 iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `max_element` requires `F` to meet the requirements of `CopyConstructible`.
- `Proj`: The type of an optional projection function. This defaults to `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 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 F, typename Proj = hpx::parallel::util::projection_identity>
util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng>>::type max_element(ExPolicy
    &&pol-
    icy,
    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 $N = \text{std}::\text{distance}(\text{first}, \text{last})$.

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.
- `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<hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. The `max_element` algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

```
template<typename FwdIter, typename Sent, typename F, typename Proj = hpx::parallel::util::projection_identity>
minmax_element_result<FwdIter, FwdIter> minmax_element(FwdIter first, Sent last, F &&f = F(),
                                                       Proj &&proj = Proj())
Finds the greatest element in the range [first, last) using the given comparison function f.
```

The assignments in the parallel `minmax_element` algorithm execute in sequential order in the calling thread.

Note Complexity: At most `max(floor(3/2*(N-1)), 0)` applications of the predicate, where `N = std::distance(first, last)`.

Template Parameters

- `FwdIter`: The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `F`: The type of the function/function object to use (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: 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.

Return The *minmax_element* algorithm returns a *minmax_element_result*<*FwdIter*, *FwdIter*>. The *minmax_element* algorithm returns a *min_max_result* consisting of an iterator to the smallest element as the *min* element and an iterator to the greatest element as the *max* element. Returns *minmax_element_result*{*first*, *first*} if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

```
template<typename Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
minmax_element_result<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> minmax_element(Rng &&rng, F &&f, Proj &&proj)
{
    = F(), Proj &&proj = Proj());
}
```

Finds the greatest element in the range [*first*, *last*) using the given comparison function *f*.

The assignments in the parallel *minmax_element* algorithm execute in sequential order in the calling thread.

Note Complexity: At most $\max(\text{floor}(3/2*(N-1)), 0)$ applications of the predicate, where $N = \text{std}:\text{distance}(\text{first}, \text{last})$.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *F*: The type of the function/function object to use (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *f*: The binary predicate which returns true if the the left argument is less than the right element. This argument is optional and defaults to *std::less*. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *minmax_element* algorithm returns a *minmax_element_result*<*hpx::traits::range_iterator*<*Rng*>::*type*, *hpx::traits::range_iterator*<*Rng*>::*type*>. The *minmax_element* algorithm returns a *min_max_result* consisting of an range iterator to the smallest element as the *min* element and an range iterator to the greatest element as the *max* element. If several elements are equivalent to the smallest element, the

iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename F, typename Proj = hpx::parallel::util::projection_identity<ExPolicy, FwdIter, Sent, F, Proj>>::type minmax_element (ExPolicy policy, FwdIter first, Sent last, F f, 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 $N = \text{std}::\text{distance}(\text{first}, \text{last})$.

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 iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for **FwdIter**.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *minmax_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *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 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 `minmax_element_result<FwdIter, FwdIter>`. The `minmax_element` algorithm returns a `min_max_result` consisting of an iterator to the smallest element as the `min` element and an iterator to the greatest element as the `max` element. Returns `minmax_element_result{first, first}` if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

```
template<typename ExPolicy, typename Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
util::detail::algorithm_result<ExPolicy, minmax_element_result<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> minmax_element(Rng first, Rng last, F f, ExPolicy exPolicy, 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 $N = \text{std}::\text{distance}(\text{first}, \text{last})$.

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.
- `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 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 `minmax_element_result<hpx::traits::range_iterator<Rng>::type, hpx::traits::range_iterator<Rng>::type>`. The `minmax_element` algorithm returns a `min_max_result` consisting of an range iterator to the smallest element as the min element and an range iterator to the greatest element as the max element. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Proj = identity> auto minmax_element(ExPolicy exec, Iter1 first, Sent1 last, Iter2 proj = Proj{}, Compare comp = less{})
```

```
util::detail::algorithm_result<ExPolicy, ranges::mismatch_result<FwdIter1, FwdIter2>>::type mismatch(ExPolicy  
    &&policy,  
    FwdIter1  
    first1,  
    FwdIter1  
    last1,  
    FwdIter2  
    first2,  
    FwdIter2  
    last2,  
    Pred  
    &&op  
    =  
    Pred(),  
    Proj1  
    &&proj1  
    =  
    Proj1(),  
    Proj2  
    &&proj2  
    =  
    Proj2())
```

Returns true if the range [first1, last1) is mismatch to the range [first2, last2), and false otherwise.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $\min(\text{last1} - \text{first1}, \text{last2} - \text{first2})$ applications of the predicate *f*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and $(\text{last1} - \text{first1}) \neq (\text{last2} - \text{first2})$ then no applications of the predicate *f* are made.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Iter1*: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent1*: The type of the source iterators used for the end of the first range (deduced).
- *Iter2*: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent2*: The type of the source iterators used for the end of the second range (deduced).
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- *Proj1*: The type of an optional projection function applied to the first range. This defaults to `util::projection_identity`
- *Proj2*: The type of an optional projection function applied to the second range. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last1*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.

- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `op`: The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The two ranges are considered mismatch if, for every iterator `i` in the range `[first1, last1)`, `*i mismatchs *(first2 + (i - first1))`. This overload of *mismatch* uses operator`==` to determine if two elements are mismatch.

Return The *mismatch* algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `bool` otherwise. The *mismatch* algorithm returns true if the elements in the two ranges are mismatch, otherwise it returns false. If the length of the range `[first1, last1)` does not mismatch the length of the range `[first2, last2)`, it returns false.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = ranges::equal_to, typename Proj1 = util::detail::algorithm_result<ExPolicy, ranges::mismatch_result<FwdIter1, FwdIter2>>::type mismatch(ExPolicy &&policy, Rng1 &&rng1, Rng2 &&rng2, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Returns std::pair with iterators to the first two non-equivalent elements.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: At most `last1 - first1` applications of the predicate *f*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1`: The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `Rng2`: The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires `Pred` to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- `Proj1`: The type of an optional projection function applied to the first range. This defaults to `util::projection_identity`
- `Proj2`: The type of an optional projection function applied to the second range. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the first sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the second sequence of elements the algorithm will be applied to.
- `op`: The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter1` and `FwdIter2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *mismatch* algorithm returns a `hpx::future<std::pair<FwdIter1, FwdIter2> >` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `std::pair<FwdIter1, FwdIter2>` otherwise. The *mismatch* algorithm returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2, last2).

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter>
util::detail::algorithm_result<ExPolicy, ranges::move_result<FwdIter1, FwdIter>>::type move (ExPolicy
    &&policy,
    FwdIter1
    iter,
    Sent1
    sent,
    FwdIter
    dest)
```

Moves the elements in the range *rng* to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for **FwdIter1**.
- **FwdIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *move* algorithm returns a *hpx::future<ranges::move_result<iterator_t<Rng>, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::move_result<iterator_t<Rng>, FwdIter2>* otherwise. The *move* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element moved.

```
template<typename ExPolicy, typename Rng, typename FwdIter>
```

```
util::detail::algorithm_result<ExPolicy, ranges::move_result<typename hpx::traits::range_traits<Rng>::iterator_type, FwdIter>
```

Moves the elements in the range *rng* to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *FwdIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *move* algorithm returns a *hpx::future<ranges::move_result<iterator_t<Rng>, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::move_result<iterator_t<Rng>, FwdIter2>* otherwise. The *move* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element moved.

namespace hpx

namespace ranges

Functions

template<typename **RandomIt**, typename **Sent**, typename **Pred**, typename **Proj**>

RandomIt nth_element (*RandomIt first, RandomIt nth, Sent last, Pred &&pred, Proj &&proj*)

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by *nth* is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new *nth* element are less than or equal to the elements after the new *nth* element.

The comparison operations in the parallel *nth_element* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear in `std::distance(first, last)` on average. $O(N)$ applications of the predicate, and $O(N \log N)$ swaps, where $N = \text{last} - \text{first}$.

Template Parameters

- **RandomIt**: The type of the source begin, *nth*, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **RandomIt**.
- **Pred**: Comparison function object which returns true if the first argument is less than the second.
- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **nth**: Refers to the iterator defining the sort partition point
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred**: Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type must be such that an object of type `randomIt` can be dereferenced and then implicitly converted to `Type`. This defaults to `std::less<>`.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. This defaults to `projection_identity`.

Return The *nth_element* algorithm returns returns **RandomIt**. The *nth_element* algorithm returns an iterator equal to *last*.

template<typename **ExPolicy**, typename **RandomIt**, typename **Sent**, typename **Pred**, typename **Proj**>

`parallel::util::detail::algorithm_result_t<ExPolicy, RandomIt> nth_element` (*ExPolicy &&policy, RandomIt first, RandomIt nth, Sent last, Pred &&pred, Proj &&proj*)

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by *nth* is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new *nth* element are less than or equal to the elements after the new *nth* element.

The comparison operations in the parallel *nth_element* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandomIt**: The type of the source begin, nth, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Pred**: Comparison function object which returns true if the first argument is less than the second.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **nth**: Refers to the iterator defining the sort partition point
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred**: Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. This defaults to *projection_identity*.

The assignments in the parallel *nth_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partition* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *parallel_task_policy* and returns *RandomIt* otherwise. The *nth_element* algorithm returns an iterator equal to *last*.

```
template<typename Rng, typename Pred, typename Proj>
hpx::traits::range_iterator_t<Rng> nth_element(Rng &&rng, hpx::traits::range_iterator_t<Rng>
                                                nth, Pred &&pred, Proj &&proj)
```

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by *nth* is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new *nth* element are less than or equal to the elements after the new *nth* element.

The comparison operations in the parallel *nth_element* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters

- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Pred**: Comparison function object which returns true if the first argument is less than the second.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **nth**: Refers to the iterator defining the sort partition point
- **pred**: Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type must be such that an object of type `randomIt` can be dereferenced and then implicitly converted to `Type`. This defaults to `std::less<>`.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. This defaults to `projection_identity`.

Return The `nth_element` algorithm returns returns `hpx::traits::range_iterator_t<Rng>`. The `nth_element` algorithm returns an iterator equal to last.

```
template<typename ExPolicy, typename Rng, typename Pred, typename Proj>
parallel::util::detail::algorithm_result_t<ExPolicy, hpx::traits::range_iterator_t<Rng>> nth_element(ExPolicy
&&pol-
icy,
Rng
&&rng,
hpx::traits::range_
nth,
Pred
&&pred,
Proj
&&proj)
```

`nth_element` is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by `nth` is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new `nth` element are less than or equal to the elements after the new `nth` element.

The comparison operations in the parallel `nth_element` invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Linear in `std::distance(first, last)` on average. $O(N)$ applications of the predicate, and $O(N \log N)$ swaps, where $N = \text{last} - \text{first}$.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Pred**: Comparison function object which returns true if the first argument is less than the second.

- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `nth`: Refers to the iterator defining the sort partition point
- `pred`: Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type must be such that an object of type `randomIt` can be dereferenced and then implicitly converted to `Type`. This defaults to `std::less<>`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `is` is invoked. This defaults to `projection_identity`.

The assignments in the parallel `nth_element` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `partition` algorithm returns a `hpx::future<hpx::traits::range_iterator_t<Rng>>` if the execution policy is of type `parallel_task_policy` and returns `hpx::traits::range_iterator_t<Rng>` otherwise. The `nth_element` algorithm returns an iterator equal to `last`.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename RandomIt, typename Sent, typename Comp, typename Proj>
RandomIt partial_sort(RandomIt first, Sent last, Comp &&comp = Comp(), Proj &&proj =
                      Proj())
```

Places the first `middle - first` elements from the range `[first, last)` as sorted with respect to `comp` into the range `[first, middle)`. The rest of the elements in the range `[middle, last)` are placed in an unspecified order.

The assignments in the parallel `partial_sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Approximately $(last - first) * \log(middle - first)$ comparisons.

Template Parameters

- `RandomIt`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `RandomIt`.
- `Comp`: The type of the function/function object to use (deduced). `Comp` defaults to `detail::less`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `middle`: Refers to the middle of the sequence of elements the algorithm will be applied to.

- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `comp`: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator. `Comp` defaults to `detail::less`.
- `proj`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.

Return The `partial_sort` algorithm returns `RandomIt`. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

```
template<typename ExPolicy, typename RandomIt, typename Sent, typename Comp, typename Proj>
parallel::util::detail::algorithm_result_t<ExPolicy, RandomIt> partial_sort(ExPolicy &&policy,
                                                               RandomIt first,
                                                               RandomIt middle,
                                                               Sent last,
                                                               Comp &&comp =
                                                               = Comp(), Proj
                                                               &&proj = Proj())
```

Places the first `middle - first` elements from the range `[first, last)` as sorted with respect to `comp` into the range `[first, middle)`. The rest of the elements in the range `[middle, last)` are placed in an unspecified order.

The application of function objects in parallel algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Approximately $(last - first) * \log(middle - first)$ comparisons.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `RandomIt`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `RandomIt`.
- `Comp`: The type of the function/function object to use (deduced). `Comp` defaults to `detail::less`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `middle`: Refers to the middle of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `comp`: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator. `Comp` defaults to `detail::less`.
- `proj`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partial_sort* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

template<typename Rng, typename Comp, typename Proj>hpx::traits::range_iterator<Rng>

Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

The assignments in the parallel *partial_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Approximately (last - first) * log(middle - first) comparisons.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Comp*: The type of the function/function object to use (deduced). *Comp* defaults to *detail::less*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *middle*: Refers to the middle of the sequence of elements the algorithm will be applied to.
- *comp*: *comp* is a callable object. The return value of the *INVOKED* operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator. *Comp* defaults to *detail::less*.
- *proj*: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Return The *partial_sort* algorithm returns *typename hpx::traits::range_iterator<Rng>::type*. It returns *last*.

template<typename ExPolicy, typename Rng, typename Comp, typename Proj>util::detail::partial_sort_fn

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and *INVOKED(comp, INVOKED(proj, *(i + n)), INVOKED(proj, *i)) == false*.

Note Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ comparisons.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Comp*: The type of the function/function object to use (deduced). *Comp* defaults to *detail::less*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **rng:** Refers to the sequence of elements the algorithm will be applied to.
- **middle:** Refers to the middle of the sequence of elements the algorithm will be applied to.
- **comp:** comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. Comp defaults to detail::less.
- **proj:** Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partial_sort* algorithm returns a *hpx::future<typename hpx::traits::range_iterator<Rng>::type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *typename hpx::traits::range_iterator<Rng>::type* otherwise. It returns *last*.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename InIter, typename Sent1, typename RandIter, typename Sent2, typename Comp, typename Proj1, typename Proj2>
partial_sort_copy_result<InIter, RandIter> partial_sort_copy(InIter first, Sent1 last, RandIter d_first, Sent2 d_last,
    Comp &&comp = Comp(),
    Proj1 &&proj1 = Proj1(),
    Proj2 &&proj2 = Proj2())
```

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [d_first, d_last). At most d_last - d_first of the elements are placed sorted to the range [d_first, d_first + n) where n is the number of elements to sort (n = min(last - first, d_last - d_first)).

The assignments in the parallel *partial_sort_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::distance(first, last)$ and $D = \text{std}::distance(d_first, d_last)$ comparisons.

Template Parameters

- **InIter:** The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1:** The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **RandIter:** The type of the destination iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent2:** The type of the destination sentinel (deduced). This sentinel type must be a sentinel for RandIter.
- **Comp:** The type of the function/function object to use (deduced). Comp defaults to detail::less.
- **Proj1:** The type of an optional projection function for the input range. This defaults to *util::projection_identity*.
- **Proj2:** The type of an optional projection function for the output range. This defaults to *util::projection_identity*.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the sentinel value denoting the end of the sequence of elements the algorithm will be applied to.
- `d_first`: Refers to the beginning of the destination range.
- `d_last`: Refers to the sentinel denoting the end of the destination range.
- `comp`: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator. This defaults to `detail::less`.
- `proj1`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate `comp` is invoked.

Return The `partial_sort_copy` algorithm returns a returns `partial_sort_copy_result<InIter, RandIter>`. The algorithm returns `{last, result_first + N}`.

```
template<typename ExPolicy, typename FwdIter, typename Sent1, typename RandIter, typename Sent2, typename parallel::util::detail::algorithm_result_t<ExPolicy, partial_sort_copy_result<FwdIter, RandIter>>> partial_sort_copy
```

Sorts some of the elements in the range $[first, last)$ in ascending order, storing the result in the range $[d_first, d_last)$. At most $d_last - d_first$ of the elements are placed sorted to the range $[d_first, d_first + n]$ where n is the number of elements to sort ($n = \min(last - first, d_last - d_first)$).

The application of function objects in parallel algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::distance(first, last)$ and $D = \text{std}::distance(d_first, d_last)$ comparisons.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in

which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1**: The type of the source sentinel (deduced).This sentinel type must be a sentinel for **FwdIter**.
- **RandIter**: The type of the destination iterators used(deduced) This iterator type must meet the requirements of an random iterator.
- **Sent2**: The type of the destination sentinel (deduced).This sentinel type must be a sentinel for **RandIter**.
- **Comp**: The type of the function/function object to use (deduced). Comp defaults to `detail::less`.
- **Proj1**: The type of an optional projection function for the input range. This defaults to `util::projection_identity`.
- **Proj1**: The type of an optional projection function for the output range. This defaults to `util::projection_identity`.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the sentinel value denoting the end of the sequence of elements the algorithm will be applied to.
- **d_first**: Refers to the beginning of the destination range.
- **d_last**: Refers to the sentinel denoting the end of the destination range.
- **comp**: comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to `detail::less`.
- **proj1**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate `comp` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `partial_sort_copy` algorithm returns a `hpx::future<partial_sort_copy_result<FwdIter, RandIter>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `partial_sort_copy_result<FwdIter, RandIter>` otherwise. The algorithm returns `{last, result_first + N}`.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent1**, typename **RandIter**, typename **Sent2**, typename

```
partial_sort_copy_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>> partial_sort_c
```

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [d_first, d_last). At most d_last - d_first of the elements are placed sorted to the range [d_first, d_first + n) where n is the number of elements to sort ($n = \min(\text{last} - \text{first}, \text{d_last} - \text{d_first})$).

The assignments in the parallel *partial_sort_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ and $D = \text{std}::\text{distance}(\text{d_first}, \text{d_last})$ comparisons.

Template Parameters

- Rng1: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a input iterator.
- Rng2: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of a random iterator.
- Comp: The type of the function/function object to use (deduced). Comp defaults to detail::less.
- Proj1: The type of an optional projection function for the input range. This defaults to *util::projection_identity*.
- Proj2: The type of an optional projection function for the output range. This defaults to *util::projection_identity*.

Parameters

- rng1: Refers to the source range.
- rng2: Refers to the destination range.
- comp: comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to detail::less.
- proj1: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.
- proj2: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate *comp* is invoked.

Return The *partial_sort_copy* algorithm returns *partial_sort_copy_result<range_iterator_t<Rng1>, range_iterator_t<Rng2>>*. The algorithm returns {last, result_first + N}.

```
template<typename ExPolicy, typename FwdIter, typename Sent1, typename RandIter, typename Sent2, typena
```

`parallel::util::detail::algorithm_result_t<ExPolicy, partial_sort_copy_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>>`

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [d_first, d_last). At most d_last - d_first of the elements are placed sorted to the range [d_first, d_first + n) where n is the number of elements to sort ($n = \min(\text{last} - \text{first}, \text{d}_\text{last} - \text{d}_\text{first})$).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ and $D = \text{std}::\text{distance}(\text{d}_\text{first}, \text{d}_\text{last})$ comparisons.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- `Rng2`: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of a random iterator.
- `Comp`: The type of the function/function object to use (deduced). `Comp` defaults to `detail::less`.
- `Proj1`: The type of an optional projection function for the input range. This defaults to `util::projection_identity`.
- `Proj2`: The type of an optional projection function for the output range. This defaults to `util::projection_identity`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the source range.
- `rng2`: Refers to the destination range.
- `comp`: `comp` is a callable object. The return value of the INVOKE operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator. This defaults to `detail::less`.
- `proj1`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate `comp` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object

of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partial_sort_copy* algorithm returns a *hpx::future<partial_sort_copy_result<range_iterator_t<Rng1>, range_iterator_t<Rng2>>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *partial_sort_copy_result<range_iterator_t<Rng1>, range_iterator_t<Rng2>>* otherwise. The algorithm returns {last, result_first + N}.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Sent, typename Pred, typename Proj>
subrange_t<FwdIter> partition(ExPolicy &&policy, Sent first, Sent last, Pred &&pred, Proj &&proj)
```

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: At most $2 * (\text{last} - \text{first})$ swaps. Exactly *last - first* applications of the predicate and projection.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an 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.

Return The *partition* algorithm returns *subrange_t*<*FwdIter*>. The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to *last*.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Pred, typename Proj>
util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter>>::type partition (ExPolicy &&policy,
                                                               FwdIter first, Sent
                                                               last, Pred &&pred,
                                                               Proj &&proj)
```

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most $2 * (\text{last} - \text{first})$ swaps. Exactly *last - first* applications of the predicate and projection.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **FwdIter**.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partition* algorithm returns a *hpx::future<subrange_t<FwdIter>>* if the execution policy is of type *parallel_task_policy* and returns *subrange_t<FwdIter>* otherwise. The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to last.

```
template<typename Rng, typename Pred, typename Proj>
subrange_t<hpx::traits::range_iterator_t<Rng>> partition(Rng &&rng, Pred &&pred, Proj &&proj)
```

Reorders the elements in the range *rng* in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs at most $2 * N$ swaps, exactly N applications of the predicate and projection, where $N = \text{std}::\text{distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *partition* algorithm returns *subrange_t<hpx::traits::range_iterator_t<Rng>>* The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to last.

```
template<typename ExPolicy, typename Rng, typename Pred, typename Proj>
```

```
util::detail::algorithm_result<ExPolicy, subrange_t<hpx::traits::range_iterator_t<Rng>>>::type partition (ExPolicy
&&policy,
Rng
&&rng,
Pred
&&pred,
Proj
&&proj)
```

Reorders the elements in the range *rng* in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked 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 $N = \text{std}:\text{distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$.

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 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<subrange_t<hpx::traits::range_iterator_t<Rng>>>* if the execution policy is of type *parallel_task_policy* and returns *sub-*

range_t<*hpx::traits::range_iterator_t*<*Rng*>> The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to last.

```
template<typename BidirIter, typename Sent, typename F, typename Proj>
subrange_t<BidirIter> stable_partition(BidirIter first, Sent last, F &&f, Proj &&proj)
```

Permutates the elements in the range [*first*, *last*] such that there exists an iterator *i* such that for every iterator *j* in the range [*first*, *i*] *INVOKE(f, INVOKE(proj, *j)) != false*, and for every iterator *k* in the range [*i*, *last*], *INVOKE(f, INVOKE(proj, *k)) == false*

The invocations of *f* in the parallel *stable_partition* algorithm invoked without an execution policy object executes in sequential order in the calling thread.

Note Complexity: At most (*last - first*) * log(*last - first*) swaps, but only linear number of swaps if there is enough extra memory Exactly *last - first* applications of the predicate and projection.

Template Parameters

- **BidirIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **BidirIter**.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **f**: Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [*first*, *last*). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have **const&**. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Return The *stable_partition* algorithm returns an iterator *i* such that for every iterator *j* in the range [*first*, *i*), *f(*j) != false* *INVOKE(f, INVOKE(proj, *j)) != false*, and for every iterator *k* in the range [*i*, *last*), *f(*k) == false* *INVOKE(f, INVOKE(proj, *k)) == false*. The relative order of the elements in both groups is preserved.

```
template<typename ExPolicy, typename BidirIter, typename Sent, typename F, typename Proj>
```

```
util::detail::algorithm_result<ExPolicy, subrange_t<BidirIter>>::type stable_partition (ExPolicy
&&policy,
BidirIter
first, Sent
last, F
&&f, Proj
&&proj)
```

Permutates the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *j)) \neq \text{false}$, and for every iterator k in the range [i, last), $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *k)) = \text{false}$

The invocations of f in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: At most $(\text{last} - \text{first}) * \log(\text{last} - \text{first})$ swaps, but only linear number of swaps if there is enough extra memory Exactly $\text{last} - \text{first}$ applications of the predicate and projection.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of f .
- **BidirIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **BidirIter**.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires F to meet the requirements of *CopyConstructible*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **f**: Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have `const&`. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate f is invoked.

The invocations of f in the parallel *stable_partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *stable_partition* algorithm returns an iterator *i* such that for every iterator *j* in the range [first, *i*), *f*(**j*) != false *INVOKE*(*f*, *INVOKE*(proj, **j*)) != false, and for every iterator *k* in the range [*i*, last), *f*(**k*) == false *INVOKE*(*f*, *INVOKE*(proj, **k*)) == false. The relative order of the elements in both groups is preserved. If the execution policy is of type *parallel_task_policy* the algorithm returns a future<> referring to this iterator.

```
template<typename Rng, typename F, typename Proj>
subrange_t<hpx::traits::range_iterator_t<Rng>> stable_partition(Rng &&rng, F &&f, Proj &&proj)
```

Permutes the elements in the range [first, last) such that there exists an iterator *i* such that for every iterator *j* in the range [first, *i*) *INVOKE*(*f*, *INVOKE*(proj, **j*)) != false, and for every iterator *k* in the range [*i*, last), *INVOKE*(*f*, *INVOKE*(proj, **k*)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked without an execution policy object executes in sequential order in the calling thread.

Note Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory Exactly *last - first* applications of the predicate and projection.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an bidirectional iterator
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *f*: Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have *const&*. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Return The *stable_partition* algorithm returns an iterator *i* such that for every iterator *j* in the range [first, *i*), *f*(**j*) != false *INVOKE*(*f*, *INVOKE*(proj, **j*)) != false, and for every iterator *k* in the range [*i*, last), *f*(**k*) == false *INVOKE*(*f*, *INVOKE*(proj, **k*)) == false. The relative order of the elements in both groups is preserved.

```
template<typename ExPolicy, typename Rng, typename F, typename Proj>
```

```
util::detail::algorithm_result<ExPolicy, subrange_t<hpx::traits::range_iterator_t<Rng>>>::type stable_partition(ExPolicy  
    &&policy,  
    Rng  
    &&&rng  
    F  
    &&f,  
    Proj  
    &&proj)
```

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *j)) \neq \text{false}$, and for every iterator k in the range [i, last), $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *k)) == \text{false}$

The invocations of f in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

Note Complexity: At most $(\text{last} - \text{first}) * \log(\text{last} - \text{first})$ swaps, but only linear number of swaps if there is enough extra memory. Exactly $\text{last} - \text{first}$ applications of the predicate and projection.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of f .
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an bidirectional iterator
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *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*: Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have *const&*. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

The invocations of f in the parallel *stable_partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range [first, i), $f(*j) \neq \text{false}$ $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *j)) \neq \text{false}$, and for every iterator k in the range [i, last), $f(*k) == \text{false}$ $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *k)) == \text{false}$. The relative order of the elements in both

groups is preserved. If the execution policy is of type *parallel_task_policy* the algorithm returns a future<> referring to this iterator.

```
template<typename FwdIter1, typename Sent, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
hpx::util::in_out_out<FwdIter1, FwdIter2, FwdIter3> partition_copy(FwdIter1 first, Sent last,
                                                               FwdIter2 dest_true,
                                                               FwdIter3 dest_false, Pred
                                                               &&pred, Proj &&proj)
```

Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *f*.

Template Parameters

- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *FwdIter2*: The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter3*: The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *dest_true*: Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- *dest_false*: Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `is` invoked.

Return The `partition_copy` algorithm returns a `partition_copy_result<FwdIter, OutIter2, OutIter3>`. The `partition_copy` algorithm returns the tuple of the source iterator `last`, the destination iterator to the end of the `dest_true` range, and the destination iterator to the end of the `dest_false` range.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename FwdIter3, typename util::detail::algorithm_result<ExPolicy, partition_copy_result<FwdIter, OutIter2, OutIter3>>::type partition_copy(ExPolicy &&policy, FwdIter first, Sent last, FwdIter dest_true, FwdIter dest_false, Pred &&pred, Proj &&proj)
```

Copies the elements in the range, defined by `[first, last)`, to two different ranges depending on the value returned by the predicate `pred`. The elements, that satisfy the predicate `pred` are copied to the range beginning at `dest_true`. The rest of the elements are copied to the range beginning at `dest_false`. The order of the elements is preserved.

The assignments in the parallel `partition_copy` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs not more than `last - first` assignments, exactly `last - first` applications of the predicate `f`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `FwdIter2`: The type of the iterator representing the destination range for the elements that satisfy the predicate `pred` (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter3`: The type of the iterator representing the destination range for the elements that don't satisfy the predicate `pred` (deduced). This iterator type must meet the requirements of an forward iterator.

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 sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *dest_true*: Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- *dest_false*: Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partition_copy* algorithm returns a *hpx::future<partition_copy_result<FwdIter, OutIter2, OutIter3>>* if the execution policy is of type *parallel_task_policy* and returns *partition_copy_result<FwdIter, OutIter2, OutIter3>* otherwise. The *partition_copy* algorithm returns the tuple of the source iterator *last*, the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

```
template<typename Rng, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
friend partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>::type partition_copy(Rng
&&rng,
FwdIter2,
dest_true,
FwdIter3,
dest_false,
Pred
&&pred,
Proj
&&proj)
```

Copies the elements in the range *rng*, to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*.

The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs not more than N assignments, exactly N applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *FwdIter2*: The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- *FwdIter3*: The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *dest_true*: Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- *dest_false*: Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *partition_copy* algorithm returns a *partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>>*. The *partition_copy* algorithm returns the tuple of the source iterator *last*, the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred**, typename **Proj**

```
friend parallel::util::detail::algorithm_result<ExPolicy, partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3> partition_copy_if(
```

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 [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *partition_copy* algorithm returns a *hpx::future<partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>>* if the execution policy is of type *parallel_task_policy* and returns *partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>* otherwise. The *partition_copy* algorithm returns the tuple of the source iterator *last*, the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename T, typename F>
util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, FwdIter first, Sent last, T init, F &&f)
>Returns GENERALIZED_SUM(f, init, *first, ..., *(first + (last - first) - 1)).
```

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *f*.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter:** The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent:** The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F:** The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T:** The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **first:** Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`. The types `Type1 Ret` must be such that an object of type `FwdIterB` can be dereferenced and then implicitly converted to any of those types.

- `init`: The initial value for the generalized sum.

The reduce operations in the parallel `copy_if` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `reduce` and `accumulate` is that the behavior of `reduce` may be non-deterministic for non-associative or non-commutative binary predicate.

Return The `reduce` algorithm returns a `hpx::future<T>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `T` otherwise. The `reduce` algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

Note `GENERALIZED_SUM(op, a1, ..., aN)` is defined as follows:

- a_1 when N is 1
- $op(GENERALIZED_SUM(op, b_1, \dots, b_K), GENERALIZED_SUM(op, b_M, \dots, b_N))$, where:
 - b_1, \dots, b_N may be any permutation of a_1, \dots, a_N and
 - $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename T>
util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, FwdIter first, Sent last, T init)
>Returns GENERALIZED_SUM(+, init, *first, ..., *(first + (last - first) - 1)).
```

The reduce operations in the parallel `reduce` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the operator `+`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `init`: The initial value for the generalized sum.

The reduce operations in the parallel `copy_if` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a `hpx::future<T>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum (applying operator`+()`) over the elements given by the input range [first, last).

Note GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename FwdIter, typename Sent>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIter>::value_type>::type reduce (ExPolicy
&&policy,
FwdIter
first,
Sent
last)
```

Returns GENERALIZED_SUM(+, T(), *first, ..., *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the operator`+()`.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a `hpx::future<T>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise (where *T* is the *value_type* of *FwdIterB*). The *reduce* algorithm returns the result of the generalized sum (applying operator`+()`) over the elements given by the input range [first, last).

Note The type of the initial value (and the result type) *T* is determined from the *value_type* of the used *FwdIterB*.

Note GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:

- b_1, \dots, b_N may be any permutation of a_1, \dots, a_N and
- $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename Rng, typename T, typename F>
util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, Rng &&rng, T init,
F &&f)
>Returns GENERALIZED_SUM(f, init, *first, ..., *(first + (last - first) - 1)).
```

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *f*.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun (const Type1 &a, const Type1 &b);
```

The signature does not need to have **const&**. The types *Type1 Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.

- **init**: The initial value for the generalized sum.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

Note *GENERALIZED_SUM*(*op*, *a*₁, ..., *a*_{*N*}) is defined as follows:

- *a*₁ when *N* is 1
- *op*(*GENERALIZED_SUM*(*op*, *b*₁, ..., *b*_{*K*}), *GENERALIZED_SUM*(*op*, *b*_{*M*}, ..., *b*_{*N*})), where:
 - b_1, \dots, b_N may be any permutation of a_1, \dots, a_N and
 - $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename Rng, typename T>
util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, Rng &&rng, T init)
>Returns GENERALIZED_SUM(+, init, *first, ..., *(first + (last - first) - 1)).
```

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the operator+().

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T**: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **init**: The initial value for the generalized sum.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

Note GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - $1 < K+1 = M \leq N$.

```
template<typename ExPolicy, typename Rng>
```

```
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_>
```

Returns GENERALIZED_SUM(+, T(), *first, ..., *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the operator+().

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

specified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Return The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise (where *T* is the *value_type* of *FwdIterB*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

Note The type of the initial value (and the result type) *T* is determined from the *value_type* of the used *FwdIterB*.

Note GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - 1 < K+1 = M <= N.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter, typename Sent, typename T, typename Proj = util::projection_identity>
subrange_t<FwdIter, Sent> remove(FwdIter first, Sent last, T const &value, Proj &&proj =
    Proj())
```

Removes all elements that are equal to *value* from the range [first, last) and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the operator==() and the projection *proj*.

Template Parameters

- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *T*: The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *value*: Specifies the value of elements to remove.
- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *remove* algorithm returns a *subrange_t<FwdIter, Sent>*. The *remove* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename T, typename Proj = util::projection_id
```

```
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter, Sent>>::type remove (ExPolicy
    &&pol-
    icy,
    FwdIter
    first,
    Sent
    last,
    T
    const
    &value,
    Proj
    &&proj =
    Proj()
)
```

Removes all elements that are equal to *value* from the range [first, last) and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the operator==() and the projection proj.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *T*: The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *value*: Specifies the value of elements to remove.
- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *remove* algorithm returns a *hpx::future<subrange_t<FwdIter, Sent>>*. The *remove* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

```
template<typename Rng, typename T, typename Proj = util::projection_identity>
subrange_t<typename hpx::traits::range_iterator<Rng>::type> remove (Rng &&rng, T const
    &value, Proj &&proj =
    Proj())
)
```

Removes all elements that are equal to *value* from the range *rng* and returns a subrange [ret, util::end(rng)), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than `util::end(rng)`

- `util::begin(rng)` assignments, exactly `util::end(rng) - util::begin(rng)` applications of the operator`==()` and the projection `proj`.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `T`: The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `value`: Specifies the value of elements to remove.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *remove* algorithm returns a `subrange_t<typename hpx::traits::range_iterator<Rng>::type>`. The *remove* algorithm returns an object `{ret, last}`, where `ret` is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

```
template<typename ExPolicy, typename Rng, typename T, typename Proj = util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<typename hpx::traits::range_iterator<Rng>::type>>::type re
```

Removes all elements that are equal to `value` from the range `rng` and and returns a subrange `[ret, util::end(rng)]`, where `ret` is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than `util::end(rng)`

- `util::begin(rng)` assignments, exactly `util::end(rng) - util::begin(rng)` applications of the operator`==()` and the projection `proj`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `T`: The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **value**: Specifies the value of elements to remove.
- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *remove* algorithm returns a *hpx::future< subrange_t<typename hpx::traits::range_iterator<Rng>::type>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove* algorithm returns the iterator to the new end of the range.

```
template<typename FwdIter, typename Sent, typename Pred, typename Proj = hpx::parallel::util::projection_identity<brange_t<FwdIter, Sent>> remove_if(FwdIter first, Sent sent, Pred &&pred, Proj &&proj = Proj())
```

Removes all elements for which predicate *pred* returns true from the range [first, last) and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *pred* and the projection *proj*.

Template Parameters

- **FwdIter**: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *FwdIter*.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible..*.
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *remove_if* algorithm returns a *subrange_t<FwdIter, Sent>*. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Pred, typename Proj = hpx::parallel::util::projection_identity<brange_t<FwdIter, Sent>> remove_if(FwdIter first, Sent sent, Pred &&pred, Proj &&proj = Proj())
```

```
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter, Sent>>::type remove_if (ExPolicy
    &&policy,
    FwdIter
    first,
    Sent
    sent,
    Pred
    &&pred,
    Proj
    &&proj
    =
    Proj())
```

Removes all elements for which predicate *pred* returns true from the range [first, last) and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *pred* and the projection *proj*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *remove_if* algorithm returns a *hpx::future<subrange_t<FwdIter, Sent>>*. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new

subrange of the values all in valid but unspecified state.

```
template<typename Rng, typename T, typename Proj = util::projection_identity>
subrange_t<typename hpx::traits::range_iterator<Rng>::type> remove_if(Rng &&rng, Pred
&&pred, Proj
&&proj = Proj())
```

Removes all elements that are equal to *value* from the range *rng* and and returns a subrange [ret, util::end(rng)), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs not more than *util::end(rng)*

- *util::begin(rng)* assignments, exactly *util::end(rng) - util::begin(rng)* applications of the operator==() and the projection *proj*.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [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.

Return The *remove_if* algorithm returns a *subrange_t<typename hpx::traits::range_iterator<Rng>::type>*. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

```
template<typename ExPolicy, typename Rng, typename Pred, typename Proj = util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<typename hpx::traits::range_iterator<Rng>::type>>::type re
```

Removes all elements that are equal to *value* from the range *rng* and and returns a subrange [ret, util::end(rng)), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *util::end(rng)*

- *util::begin(rng)* assignments, exactly *util::end(rng) - util::begin(rng)* applications of the operator==() and the projection *proj*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *remove_if* algorithm returns a *hpx::future<subrange_t< typename hpx::traits::range_iterator<Rng>::type>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

namespace hpx

namespace ranges

Functions

```
template<typename Iter, typename Sent, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_<br/>Iter replace(Iter first, Sent sent, T1 const &old_value, T2 const &new_value, Proj &&proj = Proj())
```

Replaces all elements satisfying specific criteria with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first,last) with *new_value*, when the following corresponding conditions hold: *INVOKE(proj, *i) == old_value*

Note Complexity: Performs exactly *last - first* assignments.

The assignments in the parallel *replace* algorithm execute in sequential order in the calling thread.

Template Parameters

- **Iter**: The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for **Iter**.
- **T1**: The type of the old value to replace (deduced).
- **T2**: The type of the new values to replace (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **old_value**: Refers to the old value of the elements to replace.
- **new_value**: Refers to the new value to use as the replacement.
- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *replace* algorithm returns an *Iter*.

```
template<typename Rng, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng>::type replace(Rng &&rng, T1 const &old_value, T2 const
&new_value, Proj &&proj = Proj())
Replaces all elements satisfying specific criteria with new_value in the range rng.
```

Effects: Substitutes elements referred by the iterator *it* in the range *rng* with *new_value*, when the following corresponding conditions hold: *INVOKE(proj, *i) == old_value*

Note Complexity: Performs exactly *util::end(rng) - util::begin(rng)* assignments.

The assignments in the parallel *replace* algorithm execute in sequential order in the calling thread.

Template Parameters

- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **T1**: The type of the old value to replace (deduced).
- **T2**: The type of the new values to replace (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **old_value**: Refers to the old value of the elements to replace.
- **new_value**: Refers to the new value to use as the replacement.
- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *replace* algorithm returns an *hpx::traits::range_iterator<Rng>::type*.

```
template<typename ExPolicy, typename Iter, typename Sent, typename T1, typename T2, typename Proj = hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type replace(ExPolicy &&policy, Iter
first, Sent sent, T1 const
&old_value, T2 const
&new_value, Proj &&proj
= Proj())
Replaces all elements satisfying specific criteria with new_value in the range [first, last).
```

Effects: Substitutes elements referred by the iterator *it* in the range [first,last) with new_value, when the following corresponding conditions hold: $\text{INVOKE}(\text{proj}, *i) == \text{old_value}$

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.
- **Iter**: The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *Iter*.
- **T1**: The type of the old value to replace (deduced).
- **T2**: The type of the new values to replace (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **old_value**: Refers to the old value of the elements to replace.
- **new_value**: Refers to the new value to use as the replacement.
- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise.

```
template<typename ExPolicy, typename Rng, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_identity, Rng && T1 && T2, Proj && Proj = Proj> hpx::traits::range_iterator<Rng>::type replace(ExPolicy, Rng, T1, T2, Proj, Proj, Proj)
```

Replaces all elements satisfying specific criteria with *new_value* in the range *rng*.

Effects: Substitutes elements referred by the iterator *it* in the range *rng* with *new_value*, when the

following corresponding conditions hold: `(INVOKE(proj, *i) == old_value)`

Note Complexity: Performs exactly `util::end(rng) - util::begin(rng)` assignments.

The assignments in the parallel `replace` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- `T1`: The type of the old value to replace (deduced).
- `T2`: The type of the new values to replace (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `old_value`: Refers to the old value of the elements to replace.
- `new_value`: Refers to the new value to use as the replacement.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `is` is invoked. The assignments in the parallel `replace` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Return The `replace` algorithm returns an `hpx::future<hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `hpx::traits::range_iterator<Rng>::type` otherwise.

```
template<typename Iter, typename Sent, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_if>
Iter replace_if(Iter first, Sent sent, Pred &&pred, T const &new_value, Proj &&proj = Proj())
```

Replaces all elements satisfying specific criteria (for which predicate `f` returns true) with `new_value` in the range `[first, sent]`.

Effects: Substitutes elements referred by the iterator `it` in the range `[first, sent]` with `new_value`, when the following corresponding conditions hold: `INVOKE(f, INVOKE(proj, *it)) != false`

Note Complexity: Performs exactly `sent - first` applications of the predicate.

The assignments in the parallel `replace_if` algorithm execute in sequential order in the calling thread.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Iter`: The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `Iter`.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `equal` requires `F` to meet the requirements of `CopyConstructible`. (deduced).
- `T`: The type of the new values to replace (deduced).

- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns `true` for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `Iter` can be dereferenced and then implicitly converted to `Type`.

- `new_value`: Refers to the new value to use as the replacement.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The `replace_if` algorithm returns an `Iter` It returns `last`.

```
template<typename Rng, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng>::type replace_if(Rng &&rng, Pred &&pred, T const
&new_value, Proj &&proj = Proj())
```

Replaces all elements satisfying specific criteria (for which predicate `pred` returns true) with `new_value` in the range `rng`.

Effects: Substitutes elements referred by the iterator `it` in the range `rng` with `new_value`, when the following corresponding conditions hold: `INVOKE(f, INVOKE(proj, *it)) != false`

Note Complexity: Performs exactly `util::end(rng) - util::begin(rng)` applications of the predicate.

The assignments in the parallel `replace` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `equal` requires `F` to meet the requirements of `CopyConstructible`. (deduced).
- `T`: The type of the new values to replace (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by `rng`. This is an unary predicate which returns `true` for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to `Type`.

- `new_value`: Refers to the new value to use as the replacement.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_if* algorithm returns an *hpx::traits::range_iterator<Rng>::type* It returns *last*.

```
template<typename ExPolicy, typename Iter, typename Sent, typename Pred, typename T>
    Replaces all elements satisfying specific criteria (for which predicate pred returns true) with
    new_value in the range rng.
```

Effects: Substitutes elements referred by the iterator *it* in the range *rng* with *new_value*, when the following corresponding conditions hold: *(INVOKE(f, INVOKE(proj, *it)) != false)*

Note Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Iter*: The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *Iter*.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- *T*: The type of the new values to replace (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the elements which need to 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<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy*. It returns *last*.

```
template<typename ExPolicy, typename Rng, typename Pred, typename T, typename Proj = hpx::parallel::util::proj
```

```
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type replace_if(
```

Replaces all elements satisfying specific criteria (for which predicate *pred* returns true) with *new_value* in the range *rng*.

Effects: Substitutes elements referred by the iterator *it* in the range *rng* with *new_value*, when the following corresponding conditions hold: `INVOKE(f, INVOKE(proj, *it)) != false`

Note Complexity: Performs exactly `util::end(rng) - util::begin(rng)` applications of the predicate.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*. (deduced).
- *T*: The type of the new values to replace (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by *rng*. This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *new_value*: Refers to the new value to use as the replacement.
- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_if* algorithm returns a *hpx::future<typename hpx::traits::range_iterator<Rng>::type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy*. It returns *last*.

```
template<typename InIter, typename Sent, typename OutIter, typename T1, typename T2, typename Proj = hpx::util::projection_identity>
replace_copy_result<InIter, OutIter> replace_copy(InIter first, Sent sent, OutIter dest,
                                                T1 const &old_value, T2 const &new_value, Proj &&proj = Proj())
```

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (sent - first)) either *new_value* or *(first + (it - result)) depending on whether the following corresponding condition holds: *INVOKE(proj, *(first + (i - result))) == old_value*

The assignments in the parallel *replace_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *sent - first* applications of the predicate.

Template Parameters

- *Iter*: The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *Iter*.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *T1*: The type of the old value to replace (deduced).
- *T2*: The type of the new values to replace (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *sent*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *old_value*: Refers to the old value of the elements to replace.
- *new_value*: Refers to the new value to use as the replacement.
- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *replace_copy* algorithm returns an *in_out_result<InIter, OutIter>*. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename Rng, typename OutIter, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_identity>
in_out_result<OutIter, OutIter> replace_copy(OutIter dest, T1 const &old_value, T2 const &new_value, Proj &&proj = Proj(),
```

```
replace_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter> replace_copy (Rng
&&rng,
Out-
Iter
dest,
T1
const
&old_value,
T2
const
&new_value,
Proj
&&proj
=
Proj())
```

Copies the all elements from the range `rbg` to another range beginning at `dest` replacing all elements satisfying a specific criteria with `new_value`.

Effects: Assigns to every iterator `it` in the range `[result, result + (util::end(rng) - util::begin(rng))]` either `new_value` or `*(first + (it - result))` depending on whether the following corresponding condition holds: `INVOKE(proj, *(first + (i - result))) == old_value`

The assignments in the parallel `replace_copy` algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly `util::end(rng) - util::begin(rng)` applications of the predicate.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `T1`: The type of the old value to replace (deduced).
- `T2`: The type of the new values to replace (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `old_value`: Refers to the old value of the elements to replace.
- `new_value`: Refers to the new value to use as the replacement.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The `replace_copy` algorithm returns an `in_out_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>`. The `copy` algorithm returns the pair of the input iterator `last` and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename T1, typename T2
```

```
parallel::util::detail::algorithm_result<ExPolicy, replace_copy_result<FwdIter1, FwdIter2>>::type replace_copy (ExPo
&&p
icy,
Fwd
first,
Sent
sent,
Fwd
dest,
T1
con
&old
T2
con
&new
Proj
&&p
=
Proj)
```

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator *it* in the range [result, result + (sent - first)) either *new_value* or *(first + (it - result)) depending on whether the following corresponding condition holds: $\text{INVOKE}(\text{proj}, *(\text{first} + (\text{i} - \text{result}))) == \text{old_value}$

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *sent - first* applications of the predicate.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *Iter*.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- *T1*: The type of the old value to replace (deduced).
- *T2*: The type of the new values to replace (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *sent*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *old_value*: Refers to the old value of the elements to replace.
- *new_value*: Refers to the new value to use as the replacement.
- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_copy* algorithm returns a *hpx::future<in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<FwdIter1, FwdIter2>* otherwise. The *copy* algorithm returns the pair of the forward iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng, typename FwdIter, typename T1, typename T2, typename Proj = hpx::parallel::util::detail::algorithm_result<ExPolicy, replace_copy_result<typename hpx::traits::range_iterator<Rng>::type, FwdIter>> FwdIter replace_copy(FwdIter first, FwdIter last, T1 dest, T2 new_value, ExPolicy ex_policy = ExPolicy{}, Proj proj = Proj{})
```

Copies the all elements from the range *rbg* to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator *it* in the range $[result, result + (util::end(rng) - util::begin(rng))]$ either *new_value* or $*(first + (it - result))$ depending on whether the following corresponding condition holds: $\text{(INVOKE}(proj, *(first + (i - result))) == \text{old_value})$

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly $util::end(rng) - util::begin(rng)$ applications of the predicate.

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng:** The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter:** The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T1:** The type of the old value to replace (deduced).
- **T2:** The type of the new values to replace (deduced).
- **Proj:** The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy:** The execution policy to use for the scheduling of the iterations.
- **rng:** Refers to the sequence of elements the algorithm will be applied to.
- **dest:** Refers to the beginning of the destination range.
- **old_value:** Refers to the old value of the elements to replace.
- **new_value:** Refers to the new value to use as the replacement.
- **proj:** Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *replace_copy* algorithm returns a *hpx::future<in_out_result< typename hpx::traits::range_iterator<Rng>::type, FwdIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result< typename hpx::traits::range_iterator<Rng>::type, FwdIter>>*. The *copy* algorithm returns the pair of the input iterator *last* and the forward iterator to the element in the destination range, one past the last element copied.

```
template<typename InIter, typename Sent, typename OutIter, typename Pred, typename T, typename Proj = hpx::traits::range_iterator<Rng>::type, FwdIter>>
replace_copy_if_result<InIter, OutIter> replace_copy_if(InIter first, Sent sent, OutIter dest, Pred &&pred, T const &new_value, Proj &&proj = Proj())
```

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (sent - first)) either *new_value* or *(first + (it - result)) depending on whether the following corresponding condition holds: *(INVOKE(f, INVOKE(proj, *(first + (i - result)))) != false*

The assignments in the parallel *replace_copy_if* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *sent - first* applications of the predicate.

Template Parameters

- *InIter*: The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *InIter*.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- *T*: The type of the new values to replace (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *sent*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- *new_value*: Refers to the new value to use as the replacement.
- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The `replace_copy_if` algorithm returns a `in_out_result<InIter, OutIter>`. The `replace_copy_if` algorithm returns the input iterator `last` and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename Rng, typename OutIter, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_identity<Rng>>
replace_copy_if<typename hpx::traits::range_iterator<Rng>::type, OutIter> replace_copy_if(Rng &&rng,
OutIter dest, Pred &&pred, T const &&new_value, Proj &&proj = Proj())
```

Copies the all elements from the range `rng` to another range beginning at `dest` replacing all elements satisfying a specific criteria with `new_value`.

Effects: Assigns to every iterator `it` in the range `[result, result + (util::end(rng) - util::begin(rng))]` either `new_value` or `*(first + (it - result))` depending on whether the following corresponding condition holds: `INVOKE(f, INVOKE(proj, *(first + (i - result)))) != false`

The assignments in the parallel `replace_copy_if` algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly `util::end(rng) - util::begin(rng)` applications of the predicate.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `equal` requires `Pred` to meet the requirements of `CopyConstructible`. (deduced).
- `T`: The type of the new values to replace (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by `[first, last]`. This is an unary predicate which returns `true` for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter` can be dereferenced and then implicitly converted to `Type`.

- `new_value`: Refers to the new value to use as the replacement.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The `replace_copy_if` algorithm returns an `in_out_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>`. The `replace_copy_if` algorithm returns the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename Pred, typename parallel::util::detail::algorithm_result<ExPolicy, replace_copy_if_result<FwdIter1, FwdIter2>>::type replace_copy_if(FwdIter1 first, FwdIter1 result, FwdIter1 dest, const T& new_value, Pred pred, ExPolicy ex_policy, optional<Proj> proj)
```

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (sent - first)) either *new_value* or *(first + (it - result)) depending on whether the following corresponding condition holds: `(INVOKE(f,(INVOKE(proj, *(first + (i - result)))) != false`

The assignments in the parallel `replace_copy_if` algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *sent - first* applications of the predicate.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `InIter`.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires `Pred` to meet the requirements of *CopyConstructible*. (deduced).
- `T`: The type of the new values to replace (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `sent`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns `true` for the elements which need to 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 an `hpx::future<FwdIter1, FwdIter2>`. The `replace_copy_if` algorithm returns the input iterator `last` and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng, typename FwdIter, typename Pred, typename T, typename Proj = hpx::parallel::util::detail::algorithm_result<ExPolicy, replace_copy_if_result<typename hpx::traits::range_iterator<Rng>>>::type
```

Copies the all elements from the range `rng` to another range beginning at `dest` replacing all elements satisfying a specific criteria with `new_value`.

Effects: Assigns to every iterator `it` in the range `[result, result + (util::end(rng) - util::begin(rng))]` either `new_value` or `*(first + (it - result))` depending on whether the following corresponding condition holds: `INVOKED(f, INVOKED(proj, *(first + (i - result)))) != false`

The assignments in the parallel `replace_copy_if` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `util::end(rng) - util::begin(rng)` applications of the predicate.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `equal` requires `Pred` to meet the requirements of `CopyConstructible`. (deduced).
- `T`: The type of the new values to replace (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns `true` for the elements which need to 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 an `hpx::future<in_out_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>>`. The `replace_copy_if` algorithm returns the input iterator `last` and the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename Iter, typename Sent>
Iter reverse(Iter first, Sent last)
```

Reverses the order of the elements in the range [first, last). Behaves as if applying `std::iter_swap` to every pair of iterators `first+i, (last-i) - 1` for each non-negative $i < (last-first)/2$.

The assignments in the parallel `reverse` algorithm execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between `first` and `last`.

Template Parameters

- `Iter`: The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `Iter`.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The `reverse` algorithm returns a `Iter`. It returns `last`.

```
template<typename Rng>
hpx::traits::range_iterator<Rng>::type reverse (Rng &&rng)
    Uses rng as the source range, as if using util::begin(rng) as first and ranges::end(rng) as last. Reverses
    the order of the elements in the range [first, last). Behaves as if applying std::iter_swap to every pair
    of iterators first+i, (last-i) - 1 for each non-negative i < (last-first)/2.
```

The assignments in the parallel `reverse` algorithm execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between `first` and `last`.

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a bidirectional iterator.

Parameters

- `rng`: Refers to the sequence of elements the algorithm will be applied to.

Return The `reverse` algorithm returns a `hpx::traits::range_iterator<Rng>::type`. It returns `last`.

```
template<typename ExPolicy, typename Iter, typename Sent>
parallel::util::detail::algorithm_result<ExPolicy, Iter>::type reverse (ExPolicy &&policy, Iter
    first, Sent last)
    Reverses the order of the elements in the range [first, last). Behaves as if applying std::iter_swap to
    every pair of iterators first+i, (last-i) - 1 for each non-negative i < (last-first)/2.
```

The assignments in the parallel `reverse` algorithm invoked with an execution policy object of type `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.
- `Iter`: The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `Iter`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel `reverse` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `reverse` algorithm returns a `hpx::future<Iter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `Iter` otherwise. It returns `last`.

```
template<typename ExPolicy, typename Rng>
```

`parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type reverse (ExPolicy &&policy, Rng &&rng)`

Uses `rng` as the source range, as if using `util::begin(rng)` as `first` and `ranges::end(rng)` as `last`. Reverses the order of the elements in the range `[first, last)`. Behaves as if applying `std::iter_swap` to every pair of iterators `first+i, (last-i) - 1` for each non-negative $i < (last-first)/2$.

The assignments in the parallel `reverse` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between `first` and `last`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a bidirectional iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.

The assignments in the parallel `reverse` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `reverse` algorithm returns a `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` otherwise. It returns `last`.

`template<typename Iter, typename Sent, typename OutIter>`
`reverse_copy_result<Iter, OutIter> reverse_copy (Iter first, Sent last, OutIter result)`

Copies the elements from the range `[first, last)` to another range beginning at `result` in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(result + (last - first) - 1 - i) = *(first + i)$ once for each non-negative $i < (last - first)$. If the source and destination ranges (that is, `[first, last)` and `[result, result+(last-first))` respectively) overlap, the behavior is undefined.

The assignments in the parallel `reverse_copy` algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` assignments.

Template Parameters

- `Iter`: The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- `Sent`: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `Iter`.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `result`: Refers to the begin of the destination range.

Return The *reverse_copy* algorithm returns a *reverse_copy_result*<*Iter, OutIter*>. The *reverse_copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename Rng, typename OutIter>
ranges::reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter> reverse_copy(Rng
&&rng,
Out-
Iter
re-
sult)
```

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Copies the elements from the range [first, last) to another range beginning at *result* in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(first + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [first, last) and [result, result+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a bidirectional iterator.
- *OutputIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *result*: Refers to the begin of the destination range.

Return The *reverse_copy* algorithm returns a *ranges::reverse_copy_result*< *typename hpx::traits::range_iterator<Rng>::type, OutIter*>>::type. The *reverse_copy* algorithm returns an object equal to {last, *result* + N} where N = last - first

```
template<typename ExPolicy, typename Iter, typename Sent, typename OutIter>
parallel::util::detail::algorithm_result<ExPolicy, reverse_copy_result<Iter, OutIter>>::type reverse_copy(ExPolicy
&&pol-
icy,
Iter
first,
Sent
last,
Out-
Iter
re-
sult)
```

Copies the elements from the range [first, last) to another range beginning at *result* in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(first + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [first, last) and [result, result+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter**: The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for **Iter**.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.
- **result**: Refers to the begin of the destination range.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *reverse_copy* algorithm returns a *hpx::future<reverse_copy_result<Iter, OutIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *reverse_copy_result<Iter, OutIter>* otherwise. The *reverse_copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng, typename OutIter>
util::detail::algorithm_result<ExPolicy, ranges::reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>> reverse_copy(ExPolicy exec, Rng &r, OutIter result, ranges::iterator_range<Iter> first, ranges::iterator_range<Iter> last);
```

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Copies the elements from the range [*first*, *last*] to another range beginning at *result* in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(first + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [*first*, *last*] and [*result*, *result*+(*last*-*first*)) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type

must meet the requirements of a bidirectional iterator.

- **OutputIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **result**: Refers to the begin of the destination range.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *reverse_copy* algorithm returns a *hpx::future<ranges::reverse_copy_result< type-name hpx::traits::range_iterator<Rng>::type, OutIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::reverse_copy_result< type-name hpx::traits::range_iterator<Rng>::type, OutIter>* otherwise. The *reverse_copy* algorithm returns an object equal to {last, result + N} where N = last - first

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter, typename Sent>
subrange_t<FwdIter, Sent> rotate(FwdIter first, FwdIter middle, Sent last)
```

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element *middle* becomes the first element of the new range and *middle - 1* becomes the last element.

The assignments in the parallel *rotate* algorithm execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *FwdIter*.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle**: Refers to the element that should appear at the beginning of the rotated range.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Return The *rotate* algorithm returns a *subrange_t<FwdIter, Sent>*. The *rotate* algorithm returns the iterator equal to pair(first + (last - middle), last).

```
template<typename ExPolicy, typename FwdIter, typename Sent>
```

```
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter, Sent>>::type rotate (ExPolicy
&&policy,
FwdIter
first,
FwdIter
middle,
Sent
last)
```

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element middle becomes the first element of the new range and middle - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the end iterators used (deduced). This sentinel type must be a sentinel for *FwdIter*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *middle*: Refers to the element that should appear at the beginning of the rotated range.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Return The *rotate* algorithm returns a *hpx::future<subrange_t<FwdIter, Sent>>* if the execution policy is of type *parallel_task_policy* and returns a *subrange_t<FwdIter, Sent>* otherwise. The *rotate* algorithm returns the iterator equal to pair(*first* + (*last* - *middle*), *last*).

```
template<typename Rng>
subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> rotate (Rng
&&rng,
hpx::traits::range_iterator_
mid-
dle)
```

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element middle becomes the first element of the new range and middle - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *middle*: Refers to the element that should appear at the beginning of the rotated range.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Return The *rotate* algorithm returns a *subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>>*. The *rotate* algorithm returns the iterator equal to pair(*first* + (*last* - *middle*), *last*).

template<typename **ExPolicy**, typename **Rng**>

util::detail::algorithm_result<ExPolicy, subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>>

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [*first*, *last*] in such a way that the element *middle* becomes the first element of the new range and *middle* - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *policy*: The execution policy to use for the scheduling of the iterations.
- *middle*: Refers to the element that should appear at the beginning of the rotated range.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Return The *rotate* algorithm returns a *hpx::future <subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>>* otherwise. The *rotate* algorithm returns the iterator equal to pair(*first* + (*last* - *middle*), *last*).

```
template<typename FwdIter, typename Sent, typename OutIter>
rotate_copy_result<FwdIter, OutIter> rotate_copy (FwdIter first, FwdIter middle, Sent last,
OutIter dest_first)
```

Copies the elements from the range [*first*, *last*), to another range beginning at *dest_first* in such a way, that the element *middle* becomes the first element of the new range and *middle* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for **FwdIter**.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle**: Refers to the element that should appear at the beginning of the rotated range.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

Return The *rotate_copy* algorithm returns a *rotate_copy_result*<**FwdIter**, **OutIter**>. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, rotate_copy_result<FwdIter1, FwdIter2>>::type rotate_copy (ExPolicy& policy, FwdIter first, FwdIter middle, Sent last, FwdIter dest_first)
```

Copies the elements from the range [*first*, *last*), to another range beginning at *dest_first* in such a way, that the element *middle* becomes the first element of the new range and *middle* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent**: The type of the end iterators used (deduced). This sentinel type must be a sentinel for `FwdIter`.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle**: Refers to the element that should appear at the beginning of the rotated range.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

The assignments in the parallel `rotate_copy` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `rotate_copy` algorithm returns a returns `hpx::future<rotate_copy_result<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `rotate_copy_result<FwdIter1, FwdIter2>` otherwise. The `rotate_copy` algorithm returns the output iterator to the element past the last element copied.

```
template<typename Rng, typename OutIter>
rotate_copy_result<hpx::traits::range_iterator_t<Rng>, OutIter> rotate_copy(Rng      &&rng,
                                                               hpx::traits::range_iterator_t<Rng>
                                                               middle,   OutIter
                                                               dest_first)
```

Uses `rng` as the source range, as if using `util::begin(rng)` as `first` and `ranges::end(rng)` as `last`. Copies the elements from the range `[first, last)`, to another range beginning at `dest_first` in such a way, that the element `middle` becomes the first element of the new range and `middle - 1` becomes the last element.

The assignments in the parallel `rotate_copy` algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` assignments.

Template Parameters

- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **middle**: Refers to the element that should appear at the beginning of the rotated range.
- **dest_first**: Refers to the begin of the destination range.

Return The `rotate` algorithm returns a `rotate_copy_result<hpx::traits::range_iterator_t<Rng>, OutIter>`. The `rotate_copy` algorithm returns the output iterator to the element past the last element copied.

```
template<typename ExPolicy, typename Rng, typename OutIter>
parallel::util::detail::algorithm_result<ExPolicy, rotate_copy_result<hpx::traits::range_iterator_t<Rng>, OutIter>>::type rotate(ExPolicy ex_policy, Rng rng, OutIter dest_first, range::iterator_t<Rng> middle)
```

Uses `rng` as the source range, as if using `util::begin(rng)` as `first` and `ranges::end(rng)` as `last`. Copies

the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *new_first* becomes the first element of the new range and *new_first* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **OutIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the sequence of elements the algorithm will be applied to.
- **middle**: Refers to the element that should appear at the beginning of the rotated range.
- **dest_first**: Refers to the begin of the destination range.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *rotate_copy* algorithm returns a *hpx::future<rotate_copy_result< hpx::traits::range_iterator_t<Rng>, OutIter>>* if the execution policy is of type *parallel_task_policy* and returns *rotate_copy_result< hpx::traits::range_iterator_t<Rng>, OutIter>* otherwise. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter, typename Sent, typename FwdIter2, typename Sent2, typename Pred = hpx::ranges::equal, hpx::traits::range_iterator_t<FwdIter> s_first, hpx::traits::range_iterator_t<FwdIter> s_last, hpx::traits::range_iterator_t<FwdIter2> s2_first, hpx::traits::range_iterator_t<FwdIter2> s2_last, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2()>
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **FwdIter**: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel used for the first range (deduced). This iterator type must meet the requirements of an sentinel.

- *FwdIter2*: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent2*: The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- *Proj1*: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced *FwdIter*.
- *Proj2*: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced *FwdIter2*.

Parameters

- *first*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *s_first*: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- *s_last*: Refers to the end of the sequence of elements 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.

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 Sent, typename FwdIter2, typename Sent2, typename util::detail::algorithm_result<ExPolicy, FwdIter>::type search(ExPolicy &&policy, FwdIter first, Sent last, FwdIter2 s_first, Sent2 s_last, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Searches the range $[first, last]$ for any elements in the range $[s_first, s_last]$. Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_first, s_last)$ 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 forward iterator.
- **Sent**: The type of the source sentinel used for the first range (deduced). This iterator type must meet the requirements of an sentinel.
- **FwdIter2**: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2**: The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- **Pred**: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`.
- **Proj1**: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced *FwdIter*.
- **Proj2**: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced *FwdIter2*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last**: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first**: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last**: Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op**: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
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 $[s_{\text{first}}, s_{\text{last}})$ in range $[\text{first}, \text{last})$. If the length of the subsequence $[s_{\text{first}}, s_{\text{last}})$ is greater than the length of the range $[\text{first}, \text{last})$, *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

```
template<typename Rng1, typename Rng2, typename Pred = hpx::ranges::equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

```
hpx::traits::range_iterator<Rng1>::type search (Rng1 &&rng1, Rng2 &&rng2, Pred &&op =  
Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2  
= Proj2())
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_{\text{first}}, s_{\text{last}})$ and $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- *Rng1*: The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Rng2*: The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- *Proj1*: The type of an optional projection function. This defaults to *util::projection_identity* and is applied to the elements of *Rng1*.
- *Proj2*: The type of an optional projection function. This defaults to *util::projection_identity* and is applied to the elements of *Rng2*.

Parameters

- *rng1*: Refers to the sequence of elements the algorithm will be examining.
- *rng2*: Refers to the sequence of elements the algorithm will be searching for.
- *op*: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- *proj2*: Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

Return The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [*s_first*, *s_last*) in range [first, last). If the length of the subsequence [*s_first*, *s_last*) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = hpx::ranges::equal_to, typename Pr
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type search(`

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 $S = \text{distance}(s_{\text{first}}, s_{\text{last}})$ and $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `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 FwdIter, typename FwdIter2, typename Sent2, typename Pred = hpx::ranges::equal_to, typename  
FwdIter search_n(ExPolicy &&policy, FwdIter first, std::size_t count, FwdIter2 s_first, Sent  
s_last, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 =  
Proj2())
```

Searches the range [`first`, `last`) for any elements in the range [`s_first`, `s_last`). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search_n* algorithm execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{count}$.

Template Parameters

- `FwdIter`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent2`: The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires `Pred` to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`

Parameters

- `first`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `count`: Refers to the range of elements of the first range the algorithm will be applied to.
- `s_first`: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- `s_last`: Refers to the end of the sequence of elements of the algorithm will be searching for.
- `op`: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter1* as a projection operation before the actual predicate *is* invoked.

- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced `FwdIter2` as a projection operation before the actual predicate *is* invoked.

Return The `search_n` algorithm returns `FwdIter`. The `search_n` algorithm returns an iterator to the beginning of the last subsequence $[s_first, s_last]$ in range $[first, first+count]$. If the length of the subsequence $[s_first, s_last]$ is greater than the length of the range $[first, first+count]$, `first` is returned. Additionally if the size of the subsequence is empty or no subsequence is found, `first` is also returned.

```
template<typename ExPolicy, typename FwdIter, typename FwdIter2, typename Sent2, typename Pred = hpx::rangeutil::detail::algorithm_result<ExPolicy, FwdIter>::type search_n(ExPolicy &&policy, FwdIter first, std::size_t count, FwdIter2 s_first, Sent2 s_last, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Searches the range $[first, last]$ for any elements in the range $[s_first, s_last]$. Uses a provided predicate to compare elements.

The comparison operations in the parallel `search_n` algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{count}$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent2`: The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `adjacent_find` requires `Pred` to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `count`: Refers to the range of elements of the first range the algorithm will be applied to.
- `s_first`: Refers to the beginning of the sequence of elements the algorithm will be searching for.
- `s_last`: Refers to the end of the sequence of elements of the algorithm will be searching for.
- `op`: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
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 Rng1, typename Rng2, typename Pred = hpx::ranges::equal_to, typename Proj1 = hpx::parallel::util::traits::range_iterator<Rng1>::type>
hpx::search_n(Rng1 &&rng1, std::size_t count, Rng2 &&rng2, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Searches the range [*first*, *last*) for any elements in the range [*s_first*, *s_last*). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note Complexity: at most (*S***N*) comparisons where *S* = distance(*s_first*, *s_last*) and *N* = distance(*first*, *last*).

Template Parameters

- *Rng1*: The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Rng2*: The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to<>*
- *Proj1*: The type of an optional projection function. This defaults to *util::projection_identity* and is applied to the elements of *Rng1*.
- *Proj2*: The type of an optional projection function. This defaults to *util::projection_identity* and is applied to the elements of *Rng2*.

Parameters

- *rng1*: Refers to the sequence of elements the algorithm will be examining.
- *count*: The number of elements to apply the algorithm on.
- *rng2*: Refers to the sequence of elements the algorithm will be searching for.
- *op*: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.

- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of `rng2` as a projection operation before the actual predicate `is` invoked.

Return The `search` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `task_execution_policy` and returns `FwdIter` otherwise. The `search` algorithm returns an iterator to the beginning of the first subsequence $[s_{\text{first}}, s_{\text{last}}]$ in range $[\text{first}, \text{last}]$. If the length of the subsequence $[s_{\text{first}}, s_{\text{last}}]$ is greater than the length of the range $[\text{first}, \text{last}]$, `last` is returned. Additionally if the size of the subsequence is empty `first` is returned. If no subsequence is found, `last` is returned.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = hpx::ranges::equal_to, typename Pr hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type search_
```

Searches the range $[\text{first}, \text{last}]$ for any elements in the range $[s_{\text{first}}, s_{\text{last}}]$. 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 $S = \text{distance}(s_{\text{first}}, s_{\text{last}})$ and $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `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.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename It...
```

```
util::detail::algorithm_result<ExPolicy, ranges::set_difference_result<Iter1, Iter3>>::type set_difference (ExPolicy
&&policy,
Iter1,
first1,
Sent1
last1,
Iter2
first2,
Sent2
last2,
Iter3
dest,
Pred
&&op
=
Pred(),
Proj1
&&proj1
=
Proj1(),
Proj2
&&proj2
=
Proj2())
```

Constructs a sorted range beginning at *dest* consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter1**: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2**: The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2**: The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3**: The type of the iterator representing the destination range (deduced). This iterator

type must meet the requirements of an output iterator.

- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- *Proj1*: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- *Proj2*: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last1*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *first2*: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- *last2*: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- *proj2*: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *set_difference* algorithm returns a `hpx::future<ranges::set_difference_result<Iter1, Iter3>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `ranges::set_difference_result<Iter1, Iter3>` otherwise. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Iter3, typename Pred = detail::less, typename
```

```
util::detail::algorithm_result<ExPolicy, ranges::set_difference_result<typename traits::range_iterator<Rng1>::type, Iter3>
```

Constructs a sorted range beginning at *dest* consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- *Rng1*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Rng2*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Iter3*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- *Proj1*: The type of an optional projection function applied to the first sequence. This defaults to *util::projection_identity*
- *Proj2*: The type of an optional projection function applied to the second sequence. This defaults to *util::projection_identity*

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the first sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the second sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type1` must be such that objects of type `InIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate `op` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `set_difference` algorithm returns a `hpx::future<ranges::set_difference_result<Iter1, Iter3>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `ranges::set_difference_result<Iter1, Iter3>` otherwise. where `Iter1` is `range_iterator_t<Rng1>` and `Iter2` is `range_iterator_t<Rng2>`. The `set_difference` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename It
```

```
util::detail::algorithm_result<ExPolicy, ranges::set_intersection_result<Iter1, Iter2, Iter3>>::type set_intersection(
```

Constructs a sorted range beginning at `dest` consisting of all elements present in both sorted ranges `[first1, last1)` and `[first2, last2)`. This algorithm expects both input ranges to be sorted with the given binary predicate `f`.

If some element is found m times in `[first1, last1)` and n times in `[first2, last2)`, the first `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.
- `Iter1`: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- `Sent1`: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for `Iter1`.
- `Iter2`: The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- `Sent2`: The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for `Iter2`.
- `Iter3`: The type of the iterator representing the destination range (deduced). This iterator

type must meet the requirements of an output iterator.

- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<T>`
- *Proj1*: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- *Proj2*: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last1*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *first2*: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- *last2*: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- *proj2*: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *set_intersection* algorithm returns a `hpx::future<ranges::set_intersection_result<Iter1, Iter2, Iter3>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `ranges::set_intersection_result<Iter1, Iter2, Iter3>` otherwise. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = detail::less, typename **Op** = detail::equal>

`util::detail::algorithm_result<ExPolicy, ranges::set_intersection_result<typename traits::range_iterator<Rng1>::type, typename traits::range_iterator<Rng2>::type, Pred, Proj1, Proj2>>`

Constructs a sorted range beginning at `dest` consisting of all elements present in both sorted ranges `[first1, last1)` and `[first2, last2)`. This algorithm expects both input ranges to be sorted with the given binary predicate `f`.

If some element is found m times in `[first1, last1)` and n times in `[first2, last2)`, the first `std::min(m, n)` elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where $N1$ is the length of the first sequence and $N2$ is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `Rng1`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Rng2`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Iter3`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `set_intersection` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`
- `Proj1`: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- `Proj2`: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the first sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the second sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type1` must be such that objects of type `InIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate `op` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `set_intersection` algorithm returns a `hpx::future<ranges::set_intersection_result<Iter1, Iter2, Iter3>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `ranges::set_intersection_result<Iter1, Iter2, Iter3>` otherwise. where `Iter1` is `range_iterator_t<Rng1>` and `Iter2` is `range_iterator_t<Rng2>` The `set_intersection` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename It
```

`util::detail::algorithm_result<ExPolicy, ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>>::type set_symmetr`

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.
- **Iter1:** The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1:** The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2:** The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2:** The type of the end source iterators used (deduced) representing the second sequence.

This iterator type must meet the requirements of an sentinel for *Iter2*.

- *Iter3*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- *Proj1*: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- *Proj2*: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last1*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *first2*: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- *last2*: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- *proj2*: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *set_symmetric_difference* algorithm returns a `hpx::future<ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>` otherwise. The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Iter3, typename Pred = detail::less, typename
```

```
util::detail::algorithm_result<ExPolicy, ranges::set_symmetric_difference_result<typename traits::range_iterator<Rng1>::type, typename traits::range_iterator<Rng2>::type, Pred, Proj1> set_symmetric_difference(Iter3 dest, const Iter1 first1, const Iter1 last1, const Iter1 first2, const Iter1 last2, Pred pred, Proj1 proj1);
```

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 $\text{std::abs}(m-n)$ 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 $2*(N_1 + N_2 - 1)$ comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- **ExPolicy:** The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng1:** The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2:** The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3:** The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred:** The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::less<>*
- **Proj1:** The type of an optional projection function applied to the first sequence. This defaults to *util::projection_identity*

- `Proj2`: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the first sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the second sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
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 `Inter` can be dereferenced and then implicitly converted to `Type1`

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate `op` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `set_symmetric_difference` algorithm returns a `hpx::future<ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>` otherwise. where `Iter1` is `range_iterator_t<Rng1>` and `Iter2` is `range_iterator_t<Rng2>` The `set_symmetric_difference` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename It
```

```
util::detail::algorithm_result<ExPolicy, ranges::set_union_result<Iter1, Iter2, Iter3>>::type set_union(ExPolicy
    &&policy,
    Iter1
    first1,
    Sent1
    last1,
    Iter2
    first2,
    Sent2
    last2,
    Iter3
    dest,
    Pred
    &&op
    =
    Pred(),
    Proj1
    &&proj1
    =
    Proj1(),
    Proj2
    &&proj2
    =
    Proj2())
```

Constructs a sorted range beginning at `dest` consisting of all elements present in one or both sorted ranges `[first1, last1]` and `[first2, last2]`. This algorithm expects both input ranges to be sorted with the given binary predicate `f`.

If some element is found m times in `[first1, last1]` and n times in `[first2, last2]`, then all m elements will be copied from `[first1, last1]` to `dest`, preserving order, and then exactly `std::max(n-m, 0)` elements will be copied from `[first2, last2]` to `dest`, also preserving order.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where $N1$ is the length of the first sequence and $N2$ is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*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.
- `Iter1`: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- `Sent1`: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for `Iter1`.
- `Iter2`: The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- `Sent2`: The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for `Iter2`.
- `Iter3`: The type of the iterator representing the destination range (deduced). This iterator

type must meet the requirements of an output iterator.

- *Pred*: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_union* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- *Proj1*: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- *Proj2*: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- *last1*: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- *first2*: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- *last2*: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*

- *proj1*: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- *proj2*: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *set_union* algorithm returns a `hpx::future<ranges::set_union_result<Iter1, Iter2, Iter3>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `ranges::set_union_result<Iter1, Iter2, Iter3>` otherwise. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Iter3, typename Pred = detail::less, typename
```

`util::detail::algorithm_result<ExPolicy, ranges::set_union_result<typename traits::range_iterator<Rng1>::type, typename traits::range_iterator<Rng2>::type, Pred, Proj1, Proj2>`

Constructs a sorted range beginning at `dest` consisting of all elements present in one or both sorted ranges $[first1, last1]$ and $[first2, last2]$. This algorithm expects both input ranges to be sorted with the given binary predicate f .

If some element is found m times in $[first1, last1]$ and n times in $[first2, last2]$, then all m elements will be copied from $[first1, last1]$ to `dest`, preserving order, and then exactly `std::max(n-m, 0)` elements will be copied from $[first2, last2]$ to `dest`, also preserving order.

Note Complexity: At most $2*(N1 + N2 - 1)$ comparisons, where $N1$ is the length of the first sequence and $N2$ is the length of the second sequence.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- `Rng1`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Rng2`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Iter3`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `set_union` requires `Pred` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`
- `Proj1`: The type of an optional projection function applied to the first sequence. This defaults to `util::projection_identity`
- `Proj2`: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the first sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the second sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type `Type1` must be such that objects of type `InIter` can be dereferenced and then implicitly converted to `Type1`.

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate `op` is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate `op` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `set_union` algorithm returns a `hpx::future<ranges::set_union_result<Iter1, Iter2, Iter3>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `ranges::set_union_result<Iter1, Iter2, Iter3>` otherwise. where `Iter1` is `range_iterator_t<Rng1>` and `Iter2` is `range_iterator_t<Rng2>` The `set_union` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Sent, typename Size>
FwdIter shift_left(FwdIter first, Sent last, Size n)
```

Shifts the elements in the range [first, last) by `n` positions towards the beginning of the range. For every integer `i` in $[0, \text{last} - \text{first}]$

- `n`), moves the element originally at position `first + n + i` to position `first + i`.

The assignment operations in the parallel `shift_left` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: At most $(\text{last} - \text{first}) - n$ assignments.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `Size`: The type of the argument specifying the number of positions to shift by.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `n`: Refers to the number of positions to shift.

Note The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

Return The `shift_left` algorithm returns `FwdIter`. The `shift_left` algorithm returns an iterator to the end of the resulting range.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter> shift_left (ExPolicy &&policy, FwdIter
first, Sent last, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position `first + n + i` to position `first + i`.

The assignment operations in the parallel `shift_left` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `Size`: The type of the argument specifying the number of positions to shift by.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `n`: Refers to the number of positions to shift.

The assignment operations in the parallel `shift_left` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

Return The `shift_left` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. The `shift_left` algorithm returns an iterator to the end of the resulting range.

```
template<typename Rng, typename Size>
hpx::traits::range_iterator_t<Rng> shift_left (Rng &&rng, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i.

The assignment operations in the parallel *shift_left* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- Rng: The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- Size: The type of the argument specifying the number of positions to shift by.

Parameters

- rng: Refers to the range in which the elements will be shifted.
- n: Refers to the number of positions to shift.

Note The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Return The *shift_left* algorithm returns *hpx::traits::range_iterator_t<Rng>*. The *shift_left* algorithm returns an iterator to the end of the resulting range.

```
template<typename ExPolicy, typename Rng, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng>>::type shift_left (ExPolicy
&&policy,
Rng
&&rng,
Size
n)
```

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i.

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- ExPolicy: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- Rng: The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- Size: The type of the argument specifying the number of positions to shift by.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the range in which the elements will be shifted.
- `n`: Refers to the number of positions to shift.

The assignment operations in the parallel `shift_left` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced `hpx::traits::range_iterator_t<Rng>` must meet the requirements of `MoveAssignable`.

Return The `shift_left` algorithm returns a `hpx::future<hpx::traits::range_iterator_t<Rng>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `hpx::traits::range_iterator_t<Rng>` otherwise. The `shift_left` algorithm returns an iterator to the end of the resulting range.

```
namespace hpx
```

Functions

```
template<typename FwdIter, typename Sent, typename Size>
FwdIter shift_right(FwdIter first, Sent last, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- `n + i`.

The assignment operations in the parallel `shift_right` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `Size`: The type of the argument specifying the number of positions to shift by.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `n`: Refers to the number of positions to shift.

Note The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

Return The *shift_right* algorithm returns *FwdIter*. The *shift_right* algorithm returns an iterator to the end of the resulting range.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename SizeExPolicy, FwdIter> shift_right (ExPolicy && policy, FwdIter
first, Sent last, Size n)
Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i
in [0, last - first - n), moves the element originally at position first + i to position first
• n + i.
```

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- **Size**: The type of the argument specifying the number of positions to shift by.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **n**: Refers to the number of positions to shift.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Return The *shift_right* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *shift_right* algorithm returns an iterator to the end of the resulting range.

```
template<typename Rng, typename SizeRng> shift_right (Rng && rng, Size n)
Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i
in [0, last - first - n), moves the element originally at position first + i to position first
• n + i.
```

The assignment operations in the parallel *shift_right* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- **Rng**: The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of positions to shift by.

Parameters

- **rng**: Refers to the range in which the elements will be shifted.
- **n**: Refers to the number of positions to shift.

Note The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Return The *shift_right* algorithm returns *hpx::traits::range_iterator_t<Rng>*. The *shift_right* algorithm returns an iterator to the end of the resulting range.

```
template<typename ExPolicy, typename Rng, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng>>::type shift_right (ExPolicy
&&policy,
Rng
&&rng,
Size
n)
```

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- $n + i$.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most (last - first) - n assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng**: The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size**: The type of the argument specifying the number of positions to shift by.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: Refers to the range in which the elements will be shifted.
- **n**: Refers to the number of positions to shift.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Return The *shift_right* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::traits::range_iterator_t<Rng>* otherwise. The *shift_right* algorithm returns an iterator to the end of the resulting range.

namespace hpx

namespace ranges

Functions

template<typename RandomIt, typename Sent, typename Comp, typename Proj>
RandomIt sort (RandomIt first, Sent last, Comp &&comp, Proj &&proj)

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and *INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false*.

Note Complexity: O(Nlog(N)), where N = *detail::distance(first, last)* comparisons.

comp has to induce a strict weak ordering on the values.

Template Parameters

- *RandomIt*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *RandomIt*.
- *Comp*: The type of the function/function object to use (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *comp*: *comp* is a callable object. The return value of the *INVOKE* operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- *proj*: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Return The *sort* algorithm returns *RandomIt*. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

```
template<typename ExPolicy, typename RandomIt, typename Sent, typename Comp, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, RandomIt>::type sort (ExPolicy &&policy, RandomIt first, Sent last, Comp &&comp, Proj &&proj)
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and **INVOKE**(*comp*, **INVOKE**(*proj*, *(*i + n*)), **INVOKE**(*proj*, **i*)) == false.

Note Complexity: O(Nlog(N)), where N = **detail**::**distance**(*first*, *last*) comparisons.
comp has to induce a strict weak ordering on the values.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandomIt**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **RandomIt**.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util*::*projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp**: *comp* is a callable object. The return value of the **INVOKE** operation applied to an object of type **Comp**, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *sort* algorithm returns a *hpx::future*<*RandomIt*> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

```
template<typename Rng, typename Comp, typename Proj>
hpx::traits::range_iterator<Rng>::type sort (Rng &&rng, Compare &&comp, Proj &&proj)
```

Sorts the elements in the range *rng* in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and $\text{INVOKED}(\text{comp}, \text{INVOKED}(\text{proj}, *(\text{i} + \text{n})), \text{INVOKED}(\text{proj}, *(\text{i}))) == \text{false}$.

Note Complexity: $O(N\log(N))$, where $N = \text{std}:\text{distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$ comparisons. *comp* has to induce a strict weak ordering on the values.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Comp*: The type of the function/function object to use (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *comp*: *comp* is a callable object. The return value of the *INVOKED* operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- *proj*: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Return The *sort* algorithm returns *typename hpx::traits::range_iterator<Rng>::type*. It returns *last*.

```
template<typename ExPolicy, typename Rng, typename Pred, typename Proj>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type sort (ExPolicy
&&pol-
icy,
Rng
&&rng,
Comp
&&comp,
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 $\text{INVOKED}(\text{comp}, \text{INVOKED}(\text{proj}, *(\text{i} + \text{n})), \text{INVOKED}(\text{proj}, *(\text{i}))) == \text{false}$.

Note Complexity: $O(N\log(N))$, where $N = \text{std}:\text{distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$ 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<typename hpx::traits::range_iterator<Rng> ::type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *typename hpx::traits::range_iterator<Rng> ::type* otherwise. It returns *last*.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename RandomIt, typename Sent, typename Comp, typename Proj>
RandomIt stable_sort(RandomIt first, Sent last, Comp &&comp, Proj &&proj)
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object **comp** (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and *INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false*.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.
comp has to induce a strict weak ordering on the values.

Template Parameters

- **RandomIt**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **RandomIt**.
- **Comp**: The type of the function/function object to use (deduced).
- **Proj**: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp**: **comp** is a callable object. The return value of the INVOKE operation applied to an object of type **Comp**, when contextually converted to bool, yields true if the first argument of

the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

- `proj`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.

The assignments in the parallel `stable_sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Return The `stable_sort` algorithm returns `RandomIt`. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

```
template<typename ExPolicy, typename RandomIt, typename Sent, typename Comp, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, RandomIt>::type stable_sort (ExPolicy
    &&policy,
    RandomIt first,
    Sent last, Comp
    &&comp, Proj
    &&proj)
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object `comp` (defaults to using operator<()).

A sequence is sorted with respect to a comparator `comp` and a projection `proj` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, and `INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false`.

Note Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ 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 executes the assignments.
- `RandomIt`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `RandomIt`.
- `Comp`: The type of the function/function object to use (deduced).
- `Proj`: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `comp`: `comp` is a callable object. The return value of the `INVOKE` operation applied to an object of type `Comp`, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.
- `proj`: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate `comp` is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in

unspecified threads, and indeterminately sequenced within each thread.

Return The *stable_sort* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

```
template<typename Rng, typename Comp, typename Proj>
hpx::traits::range_iterator<Rng>::type stable_sort (Rng &&rng, Compare &&comp, Proj
&&proj)
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and *INVOKED(comp, INVOKED(proj, *(i + n)), INVOKED(proj, *i)) == false*.

Note Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ comparisons.
comp has to induce a strict weak ordering on the values.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Comp*: The type of the function/function object to use (deduced).
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *comp*: *comp* is a callable object. The return value of the *INVOKED* operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- *proj*: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

The assignments in the parallel *stable_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Return The *stable_sort* algorithm returns *typename hpx::traits::range_iterator<Rng>::type*. It returns *last*.

```
template<typename ExPolicy, typename Rng, typename Pred, typename Proj>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type stable_sort (ExPolicy
&&policy,
Rng
&&rng
Comp
&&comp
Proj&&proj)
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i*

pointing to the sequence and every non-negative integer n such that $i + n$ is a valid iterator pointing to an element of the sequence, and $\text{INVOKED}(\text{comp}, \text{INVOKED}(\text{proj}, *(\text{i} + \text{n})), \text{INVOKED}(\text{proj}, *\text{i})) == \text{false}$.

Note Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ 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.
- **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 `INVOKED` operation applied to an object of type **Comp**, when contextually converted to `bool`, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that **comp** will not apply any non-constant function through the dereferenced iterator.
- **proj**: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate **comp** is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *stable_sort* algorithm returns a `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `typename hpx::traits::range_iterator<Rng>::type` otherwise. It returns *last*.

namespace hpx

namespace ranges

Functions

```
template<typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename Pred, typename Proj1, typename Proj2>
bool starts_with(FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, Pred &&pred, Proj1 &&proj1, Proj2 &&proj2)
```

Checks whether the second range defined by [first1, last1) matches the prefix of the first range defined by [first2, last2)

The assignments in the parallel *starts_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear: at most $\min(N_1, N_2)$ applications of the predicate and both projections.

Template Parameters

- **Iter1**: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.

- **Sent1**: The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2**: The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- **Sent2**: The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred**: The binary predicate that compares the projected elements.
- **Proj1**: The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj1**: The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **first1**: Refers to the beginning of the source range.
- **last1**: Sentinel value referring to the end of the source range.
- **first2**: Refers to the beginning of the destination range.
- **last2**: Sentinel value referring to the end of the destination range.
- **pred**: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Return The *starts_with* algorithm returns *bool*. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type starts_with(ExPolicy
&&policy,
FwdIter1 first1,
Sent1 last1,
FwdIter2 first2,
Sent2 last2,
Pred &&pred,
Proj1 &&proj1,
Proj2 &&proj2)
```

Checks whether the second range defined by [first1, last1) matches the prefix of the first range defined by [first2, last2)

The assignments in the parallel *starts_with* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1**: The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **FwdIter2**: The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2**: The type of the end destination iterators used (deduced). This iterator type must meet

the requirements of an sentinel for Iter2.

- **Pred**: The binary predicate that compares the projected elements.
- **Proj1**: The type of an optional projection function for the source range. This defaults to `util::projection_identity`
- **Proj1**: The type of an optional projection function for the destination range. This defaults to `util::projection_identity`

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the source range.
- **last1**: Sentinel value referring to the end of the source range.
- **first2**: Refers to the beginning of the destination range.
- **last2**: Sentinel value referring to the end of the destination range.
- **pred**: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel `starts_with` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `starts_with` algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `bool` otherwise. The `starts_with` algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

```
template<typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
bool starts_with(Rng1 &&rng1, Rng2 &&rng2, Pred &&pred, Proj1 &&proj1, Proj2 &&proj2)
```

Checks whether the second range `rng2` matches the prefix of the first range `rng1`.

The assignments in the parallel `starts_with` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear: at most $\min(N1, N2)$ applications of the predicate and both projections.

Template Parameters

- **Rng1**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2**: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred**: The binary predicate that compares the projected elements.
- **Proj1**: The type of an optional projection function for the source range. This defaults to `util::projection_identity`
- **Proj1**: The type of an optional projection function for the destination range. This defaults to `util::projection_identity`

Parameters

- **rng1**: Refers to the source range.
- **rng2**: Refers to the destination range.
- **pred**: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each of the ele-

ments in the destination range as a projection operation before the actual predicate *is* invoked.

Return The *starts_with* algorithm returns *bool*. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type starts_with(ExPolicy
    &&policy,
    Rng1 &&rng1,
    Rng2 &&rng2,
    Pred &&pred,
    Proj1 &&proj1,
    Proj2 &&proj2)
```

Checks whether the second range *rng2* matches the prefix of the first range *rng1*.

The assignments in the parallel *starts_with* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2**: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred**: The binary predicate that compares the projected elements.
- **Proj1**: The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj2**: The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng1**: Refers to the source range.
- **rng2**: Refers to the destination range.
- **pred**: Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the two ranges projected by *proj1* and *proj2* respectively.
- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *starts_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *starts_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

namespace hpx

namespace ranges

Functions

```
template<typename InIter1, typename Sent1, typename InIter2, typename Sent2>
swap_ranges_result<InIter1, InIter2> swap_ranges (InIter1 first1, Sent1 last1, InIter2 first2,
                                                    Sent2 last2)
```

Exchanges elements between range [first1, last1) and another range starting at first2.

The swap operations in the parallel *swap_ranges* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first1* and *last1*

Template Parameters

- *InIter1*: The type of the first range of iterators to swap (deduced).
- *Sent1*: The type of the first sentinel (deduced). This sentinel type must be a sentinel for *InIter1*.
- *InIter2*: The type of the second range of iterators to swap (deduced).
- *Sent2*: The type of the second sentinel (deduced). This sentinel type must be a sentinel for *InIter2*.

Parameters

- *first1*: Refers to the beginning of the sequence of elements for the first range.
- *last1*: Refers to sentinel value denoting the end of the sequence of elements for the first range.
- *first2*: Refers to the beginning of the sequence of elements for the second range.
- *last2*: Refers to sentinel value denoting the end of the sequence of elements for the second range.

Return The *swap_ranges* algorithm returns *swap_ranges_result*<*InIter1*, *InIter2*>. The *swap_ranges* algorithm returns *in_in_result* with the first element as the iterator to the element past the last element exchanged in range beginning with *first1* and the second element as the iterator to the element past the last element exchanged in the range beginning with *first2*.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2>
parallel::util::detail::algorithm_result<ExPolicy, swap_ranges_result<FwdIter1, FwdIter2>>::type swap_ranges (ExPolicy
&&policy,
FwdIter1,
first1,
Sent1,
last1,
FwdIter2,
first2,
Sent2,
last2)
```

Exchanges elements between range [first1, last1) and another range starting at first2.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first1* and *last1*

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the first range of iterators to swap (deduced).

- `Sent1`: The type of the first sentinel (deduced). This sentinel type must be a sentinel for `FwdIter1`.
- `FwdIter2`: The type of the second range of iterators to swap (deduced).
- `Sent2`: The type of the second sentinel (deduced). This sentinel type must be a sentinel for `FwdIter2`.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the sequence of elements for the first range.
- `last1`: Refers to sentinel value denoting the end of the sequence of elements for the first range.
- `first2`: Refers to the beginning of the sequence of elements for the second range.
- `last2`: Refers to sentinel value denoting the end of the sequence of elements for the second range.

The swap operations in the parallel `swap_ranges` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `swap_ranges` algorithm returns a `hpx::future<swap_ranges_result<FwdIter1, FwdIter2>>` if the execution policy is of type `parallel_task_policy` and returns `FwdIter2` otherwise. The `swap_ranges` algorithm returns `in_in_result` with the first element as the iterator to the element past the last element exchanged in range beginning with `first1` and the second element as the iterator to the element past the last element exchanged in the range beginning with `first2`.

```
template<typename Rng1, typename Rng2>
swap_ranges_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>> swap_ranges (Rng1
&&rng1
Rng2
&&rng2)
```

Exchanges elements between range `[first1, last1]` and another range starting at `first2`.

The swap operations in the parallel `swap_ranges` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between `first1` and `last1`

Template Parameters

- `Rng1`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Rng2`: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- `rng1`: Refers to the sequence of elements of the first range.
- `rng2`: Refers to the sequence of elements of the second range.

Return The `swap_ranges` algorithm returns `swap_ranges_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>`. The `swap_ranges` algorithm returns `in_in_result` with the first element as the iterator to the element past the last element exchanged in range beginning with `first1` and the second element as the iterator to the element past the last element exchanged in the range beginning with `first2`.

```
template<typename ExPolicy, typename Rng1, typename Rng2>
```

parallel::util::detail::algorithm_result<ExPolicy, swap_ranges_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>>

Exchanges elements between range [first1, last1) and another range starting at *first2*.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first1* and *last1*

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng1*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Rng2*: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng1*: Refers to the sequence of elements of the first range.
- *rng2*: Refers to the sequence of elements of the second range.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *swap_ranges* algorithm returns a *hpx::future<swap_ranges_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>>* if the execution policy is of type *parallel_task_policy* and returns *swap_ranges_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>*. otherwise. The *swap_ranges* algorithm returns *in_in_result* with the first element as the iterator to the element past the last element exchanged in range beginning with *first1* and the second element as the iterator to the element past the last element exchanged in the range beginning with *first2*.

namespace hpx

namespace ranges

Functions

```
template<typename ExPolicy, typename Rng, typename OutIter, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, ranges::unary_transform_result<typename hpx::traits::range_iterator<Rng>::type>>
```

Applies the given function f to the given range rng and stores the result in another range, beginning at $dest$.

The invocations of f in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly $\text{size}(rng)$ 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 *range_iterator*<*Rng*>::type can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *OutIter* can be dereferenced and assigned a value of type *Ret*.

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate f is invoked.

The invocations of f in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *transform* algorithm returns a *hpx::future<ranges::unary_transform_result<range_iterator<Rng>::type, OutIter>* > if the execution policy is of type *parallel_task_policy* and returns *ranges::unary_transform_result<range_iterator<Rng>::type, OutIter>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename F, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, ranges::unary_transform_result<FwdIter1, FwdIter2>>::type transform
```

Applies the given function *f* to the given range *rng* and stores the result in another range, beginning at *dest*.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *size(rng)* applications of *f*

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- *FwdIter1*: The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent1*: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for *FwdIter1*.
- *FwdIter2*: The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of an forward iterator.
- *F*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- *last*: Refers to the end of the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *f*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

The signature does not need to have `const&`. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `Ret` must be such that an object of type `FwdIter2` can be dereferenced and assigned a value of type `Ret`.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate `f` is invoked.

The invocations of `f` in the parallel `transform` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `transform` algorithm returns a `hpx::future<ranges::unary_transform_result<FwdIter1, FwdIter2> >` if the execution policy is of type `parallel_task_policy` and returns `ranges::unary_transform_result<FwdIter1, FwdIter2>` otherwise. The `transform` algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename parallel::util::detail::algorithm_result<ExPolicy, ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>>::type t
```

Applies the given function `f` to pairs of elements from two ranges: one defined by `rng` and the other beginning at `first2`, and stores the result in another range, beginning at `dest`.

The invocations of `f` in the parallel `transform` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Exactly `size(rng)` applications of `f`

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of `f`.
- `FwdIter1`: The type of the source iterators for the first range used (deduced). This iterator

type must meet the requirements of an forward iterator.

- **Sent1**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for *FwdIter1*.
- **FwdIter2**: The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for *FwdIter2*.
- **FwdIter3**: The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj1**: The type of an optional projection function to be used for elements of the first sequence. This defaults to *util::projection_identity*
- **Proj2**: The type of an optional projection function to be used for elements of the second sequence. This defaults to *util::projection_identity*

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first1**: Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1**: Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2**: Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2**: Refers to the end of the second sequence of elements the algorithm will be applied to.
- **dest**: Refers to the beginning of the destination range.
- **f**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have **const&**. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter3* can be dereferenced and assigned a value of type *Ret*.

- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *f* is invoked.
- **proj2**: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *f* is invoked.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *transform* algorithm returns a *hpx::future<ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>>* if the execution policy is of type *parallel_task_policy* and returns *ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename InIter, typename Sent, typename OutIter, typename T, typename BinOp, typename UnOp>
transform_exclusive_scan_result<InIter, OutIter> transform_exclusive_scan(InIter first,
                                                               Sent last,
                                                               OutIter dest,
                                                               T init,
                                                               BinOp &&bi-
                                                               nary_op,
                                                               UnOp &&unary_op)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *dest*: Refers to the beginning of the destination range.
- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [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.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The *transform_exclusive_scan* algorithm returns *transform_exclusive_scan_result<InIter, OutIter>*. The *transform_exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ when N is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*_M, ..., *a*_N) where 1 < K+1 = M <= N.

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename T, typename B>
parallel::util::detail::algorithm_result<ExPolicy, transform_exclusive_result<FwdIter1, FwdIter2>>::type transform_exclusive_scan(FwdIter1 first, FwdIter1 result, FwdIter2 last, Sent sent, ExPolicy exPolicy, Op op, Conv conv, T init)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*(first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: O(last - first) applications of the predicates *op* and *conv*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will

be applied.

- `dest`: Refers to the beginning of the destination range.
- `conv`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

- `init`: The initial value for the generalized sum.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel `transform_exclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The `transform_exclusive_scan` algorithm returns a `hpx::future<transform_exclusive_result<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `transform_exclusive_result<FwdIter1, FwdIter2>` otherwise. The `transform_exclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- a_1 when $N = 1$
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_{M+1}, \dots, a_N))$ where $1 < K+1 = M \leq N$.

The behavior of `transform_exclusive_scan` may be non-deterministic for a non-associative predicate.

```
template<typename Rng, typename O, typename T, typename BinOp, typename UnOp>
transform_exclusive_scan_result<traits::range_iterator_t<Rng>, O> transform_exclusive_scan(Rng
&&rng,
O
dest,
T
init,
BinOp
&&bi-
nary_op,
UnOp
&&unary_op)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), \dots, conv(*(first + (i - result) - 1)))`.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(\text{last} - \text{first})$ applications of the predicates *op* and *conv*.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *O*: The type of the iterator representing the destination range (deduced).
- *Conv*: The type of the unary function object used for the conversion operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [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.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The *transform_exclusive_scan* algorithm returns a returns *transform_exclusive_scan_result*<traits::range_iterator_t<*Rng*>, *O*>. The *transform_exclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, a1, ..., 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 Rng, typename O, typename T, typename BinOp, typename UnOp
```

`parallel::util::detail::algorithm_result<ExPolicy, transform_exclusive_scan_result<traits::range_iterator_t<Rng>, O>>::typ`

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), \dots , conv(* $(first + (i - result) - 1)$)).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `O`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator. This iterator type must meet the requirements of an forward iterator.
- `Conv`: The type of the unary function object used for the conversion operation.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).
- `Op`: The type of the binary function object used for the reduction operation.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `conv`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by $[first, last)$. This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

- `init`: The initial value for the generalized sum.
- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects

passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The *transform_exclusive_scan* algorithm returns a *hpx::future<transform_exclusive_scan_result<traits::range_iterator_t<Rng>, O>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_exclusive_scan_result<traits::range_iterator_t<Rng>, O>* otherwise. The *transform_exclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ when N is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*_M, ..., *a*_N)) where 1 < K+1 = M <= N.

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

namespace hpx

namespace ranges

Functions

```
template<typename InIter, typename Sent, typename OutIter, typename BinOp, typename UnOp>
transform_inclusive_scan_result<InIter, OutIter> transform_inclusive_scan(InIter first,
                                                               Sent last,
                                                               OutIter dest, BinOp
                                                               &&bi-
                                                               nary_op,
                                                               UnOp
                                                               &&unary_op)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(*op*, conv(*first), ..., conv(*(first + (i - result))))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: O(last - first) applications of the predicates *op* and *conv*.

Template Parameters

- *InIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- *OutIter*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- *Op*: The type of the binary function object used for the reduction operation.
- *Conv*: The type of the unary function object used for the conversion operation.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest**: Refers to the beginning of the destination range.
- **op**: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- **conv**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

Return The `transform_inclusive_scan` algorithm returns `transform_inclusive_scan_result<InIter, OutIter>`. The `transform_inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- a_1 when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_{M+1}, \dots, a_N))$ where $1 < K+1 = M \leq N$.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename BinOp, typename parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_result<FwdIter1, FwdIter2>>::type transform_inclusive_scan
```

Assigns through each iterator i in [result, result + (last - first)) the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), ..., conv(*(first + (i - result))))`.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(\text{last} - \text{first})$ 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.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Op*: The type of the binary function object used for the reduction operation.
- *Conv*: The type of the unary function object used for the conversion operation.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *dest*: Refers to the beginning of the destination range.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *R* must be such that an object of this type can be implicitly converted to *T*.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The *transform_inclusive_scan* algorithm returns a *hpx::future<transform_inclusive_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_inclusive_result<FwdIter1, FwdIter2>* otherwise. The *transform_inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ 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_inclusive_scan may be non-deterministic for a non-associative predicate.

```
template<typename Rng, typename O, typename BinOp, typename UnOp>
transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O> transform_inclusive_scan(Rng
&&rng,
O
dest,
BinOp
&&bi-
nary_op,
UnOp
&&unary_op)
```

Assigns through each iterator i in $[result, result + (last - first)]$ the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), ..., conv(*(first + (i - result))))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *O*: The type of the iterator representing the destination range (deduced).
- *Op*: The type of the binary function object used for the reduction operation.
- *Conv*: The type of the unary function object used for the conversion operation.

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *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.

- *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*.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges $[first, last]$ or $[result, result + (last - first)]$.

Return The *transform_inclusive_scan* algorithm returns a returns *transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>*. The *transform_inclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN)) where $1 < K+1 = M \leq N$.

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

```
template<typename ExPolicy, typename Rng, typename O, typename BinOp, typename UnOp>
parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type
```

Assigns through each iterator i in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), ..., conv(*(first + (i - result))))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *O*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator. This iterator type must meet the requirements of an forward iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *Op*: The type of the binary function object used for the reduction operation.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun (const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

The reduce operations in the parallel `transform_inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

Return The `transform_inclusive_scan` algorithm returns a `hpx::future<transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>` otherwise. The `transform_inclusive_scan` algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- $a1$ when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \dots, aN))$ where $1 < K+1 = M \leq N$.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

```
template<typename InIter, typename Sent, typename OutIter, typename BinOp, typename UnOp, typename T>
transform_inclusive_scan_result<InIter, OutIter> transform_inclusive_scan(InIter first,
                                                               Sent last,
                                                               OutIter dest, BinOp
                                                               &&bin-
                                                               nary_op,
                                                               UnOp
                                                               &&unary_op,
                                                               T init)
```

Assigns through each iterator i in `[result, result + (last - first))` the value of `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), \dots, conv(*(first + (i - result))))`.

The reduce operations in the parallel `transform_inclusive_scan` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates `op` and `conv`.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter`.
- `OutIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- `Conv`: The type of the unary function object used for the conversion operation.
- `Op`: The type of the binary function object used for the reduction operation.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `dest`: Refers to the beginning of the destination range.
- `conv`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- `init`: The initial value for the generalized sum.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The `transform_inclusive_scan` algorithm returns `transform_inclusive_scan_result<InIter, OutIter>`. The `transform_inclusive_scan` algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- a_1 when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_{M+1}, \dots, a_N))$ where $1 < K+1 = M \leq N$.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename BinOp, typename parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_result<FwdIter1, FwdIter2>>::type transform_inclusive_scan
```

Assigns through each iterator i in [result, result + (last - first)) the value of `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), \dots, conv(*first + (i - result)))).`

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *Op*: The type of the binary function object used for the reduction operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *dest*: Refers to the beginning of the destination range.
- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *R* must be such that an object of this type can be implicitly converted to *T*.

- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- *init*: The initial value for the generalized sum.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The *transform_inclusive_scan* algorithm returns a *hpx::future<transform_inclusive_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_inclusive_result<FwdIter1, FwdIter2>* otherwise. The *transform_inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and

an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_N) is defined as:

- *a*₁ when N is 1
- *op*(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*₁, ..., *a*_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*_M, ..., *a*_N) where 1 < K+1 = M <= N.

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

```
template<typename Rng, typename O, typename BinOp, typename UnOp, typename T>
transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O> transform_inclusive_scan(Rng
&&rng,
O
dest,
BinOp
&&bi-
nary_op,
UnOp
&&unary_op,
T
init)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*(first + (i - result))))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *O*: The type of the iterator representing the destination range (deduced).
- *Conv*: The type of the unary function object used for the conversion operation.
- *Op*: The type of the binary function object used for the reduction operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *R* must be such that an object of this type can be implicitly converted to *T*.

- *op*: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- *init*: The initial value for the generalized sum.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The *transform_inclusive_scan* algorithm returns a returns *transform_inclusive_scan_result*<traits::range_iterator_t<Rng>, O>. The *transform_inclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

Note GENERALIZED_NONCOMMUTATIVE_SUM(*op*, a₁, ..., a_N) is defined as:

- a₁ when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(*op*, a₁, ..., a_K), GENERALIZED_NONCOMMUTATIVE_SUM(*op*, a_M, ..., a_N) where 1 < K+1 = M <= N.

The behavior of *transform_inclusive_scan* may be non-deterministic for a non-associative predicate.

```
template<typename ExPolicy, typename Rng, typename O, typename BinOp, typename UnOp, typename T>
parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: O(last - first) applications of the predicates *op* and *conv*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *O*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator. This iterator type must meet the requirements of an forward iterator.
- *Conv*: The type of the unary function object used for the conversion operation.
- *Op*: The type of the binary function object used for the reduction operation.
- *T*: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *dest*: Refers to the beginning of the destination range.
- *conv*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

- `op`: Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Ret` must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- `init`: The initial value for the generalized sum.

The reduce operations in the parallel `transform_inclusive_scan` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

Return The `transform_inclusive_scan` algorithm returns a `hpx::future<transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>` otherwise. The `transform_inclusive_scan` algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

Note `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- `a1` when `N` is 1
- `op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN))` where $1 < K+1 = M \leq N$.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

```
namespace hpx
```

Functions

```
template<typename ExPolicy, typename Rng, typename T, typename Reduce, typename Convert>
util::detail::algorithm_result<ExPolicy, T>::type transform_reduce(ExPolicy &&policy, Rng
&&rng, T init, Reduce &&red_op, Convert
&&conv_op)
```

Returns `GENERALIZED_SUM(red_op, init, conv_op(*first), ..., conv_op(*(first + (last - first) - 1)))`.

The reduce operations in the parallel `transform_reduce` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicates `red_op` and `conv_op`.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `copy_if` requires `F` to meet the requirements of `CopyConstructible`.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).
- `Reduce`: The type of the binary function object used for the reduction operation.
- `Convert`: The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `init`: The initial value for the generalized sum.
- `red_op`: Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of `conv_op`. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1`, `Type2`, and `Ret` must be such that an object of a type as returned from `conv_op` can be implicitly converted to any of those types.

- `conv_op`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `Iter` can be dereferenced and then implicitly converted to `Type`. The type `R` must be such that an object of this type can be implicitly converted to `T`.

The reduce operations in the parallel `transform_reduce` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between `transform_reduce` and `accumulate` is that the behavior of `transform_reduce` may be non-deterministic for non-associative or non-commutative binary predicate.

Return The `transform_reduce` algorithm returns a `hpx::future<T>` if the execution policy is of type `parallel_task_policy` and returns `T` otherwise. The `transform_reduce` algorithm returns the result of the generalized sum over the values returned from `conv_op` when applied to the elements given by the input range [first, last).

Note `GENERALIZED_SUM(op, a1, ..., aN)` is defined as follows:

- a_1 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 Rng1, typename FwdIter2, typename T>
util::detail::algorithm_result<ExPolicy, T>::type transform_reduce (ExPolicy &&policy, Rng1
&&rng1, FwdIter2 first2, T
init)
```

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(\text{last} - \text{first})$ 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.
- **Rng1**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter2**: The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to be used as return) values (deduced).

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng1**: Refers to the sequence of elements the algorithm will be applied to.
- **first2**: Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init**: The initial value for the sum.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise.

```
template<typename ExPolicy, typename Rng1, typename FwdIter2, typename T, typename Reduce, typename Convert>
util::detail::algorithm_result<ExPolicy, T>::type transform_reduce (ExPolicy &&policy, Rng1
&&rng1, FwdIter2 first2,
T init, Reduce &&red_op,
Convert &&conv_op)
```

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: $O(last - first)$ applications of the predicate *op2*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng1*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *FwdIter2*: The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *T*: The type of the value to be used as return) values (deduced).
- *Reduce*: The type of the binary function object used for the multiplication operation.
- *Convert*: The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng1*: Refers to the sequence of elements the algorithm will be applied to.
- *first2*: Refers to the beginning of the second sequence of elements the result will be calculated with.
- *init*: The initial value for the sum.
- *red_op*: Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *op2*. This is a binary predicate. The signature of this predicate should be equivalent to:

```
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.

```
namespace hpx
```

```
    namespace ranges
```

Functions

```
template<typename InIter, typename Sent1, typename FwdIter, typename Sent2>
hpx::parallel::util::in_out_result<InIter, FwdIter> uninitialized_copy(InIter first1, Sent1
last1, FwdIter first2,
Sent2 last2)
```

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly last - first assignments.

Template Parameters

- InIter: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- Sent1: The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- FwdIter: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- Sent2: The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters

- first1: Refers to the beginning of the sequence of elements that will be copied from
- last1: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied
- first2: Refers to the beginning of the destination range.
- last2: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Return The *uninitialized_copy* algorithm returns an *in_out_result<InIter, FwdIter>*. The *uninitialized_copy* algorithm returns an input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2>
parallel::util::detail::algorithm_result<ExPolicy, parallel::util::in_out_result<FwdIter1, FwdIter2>>::type uninitialized_copy
```

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Sent1*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- *Sent2*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter2*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements that will be copied from
- *last1*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *first2*: Refers to the beginning of the destination range.
- *last2*: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_copy* algorithm returns a *hpx::future<in_out_result<InIter, FwdIter>>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<InIter, FwdIter>* otherwise. The *uninitialized_copy* algorithm returns an input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename Rng1, typename Rng2>
hpx::parallel::util::in_out_result<typename hpx::traits::range_traits<Rng1>::iterator_type, typename hpx::traits::range_
```

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *Rng1*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Rng2*: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- *rng1*: Refers to the range from which the elements will be copied from
- *rng2*: Refers to the range to which the elements will be copied to

Return The *uninitialized_copy* algorithm returns an *in_out_result*<typename hpx::traits::range_traits<Rng1> ::iterator_type, typename hpx::traits::range_traits<Rng2> ::iterator_type>. The *uninitialized_copy* algorithm returns an input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename Rng1, typename Rng2>
```

```
parallel::util::detail::algorithm_result<ExPolicy, hpx::parallel::util::in_out_result<typename hpx::traits::range_traits<Rng1> ::iterator_type, typename hpx::traits::range_traits<Rng2> ::iterator_type>> uninitialized_copy(
```

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng1*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *Rng2*: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng1*: Refers to the range from which the elements will be copied from
- *rng2*: Refers to the range to which the elements will be copied to

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_copy* algorithm returns a *hpx::future*<*in_out_result*<*InIter*, *FwdIter*>>, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result*< typename hpx::traits::range_traits<Rng1>::iterator_type , typename hpx::traits::range_traits<Rng2>::iterator_type> otherwise. The *uninitialized_copy* algorithm returns the input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename InIter, typename Size, typename FwdIter, typename Sent2>
```

```
hpx::parallel::util::in_out_result<InIter, FwdIter> uninitialized_copy_n(
```

<i>InIter</i> <i>first1</i> ,	<i>Size</i> <i>count</i> ,	<i>FwdIter</i> <i>first2</i> ,	<i>Sent2</i> <i>last2</i>)
-------------------------------	----------------------------	--------------------------------	-----------------------------

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy ob-

ject of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **InIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size**: The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **FwdIter**.

Parameters

- **first1**: Refers to the beginning of the sequence of elements that will be copied from
- **count**: Refers to the number of elements starting at *first* the algorithm will be applied to.
- **first2**: Refers to the beginning of the destination range.
- **last2**: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Return The *uninitialized_copy_n* algorithm returns *in_out_result<InIter, FwdIter>*. The *uninitialized_copy_n* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2, typename Sent2>
parallel::util::detail::algorithm_result<ExPolicy, parallel::util::in_out_result<FwdIter1, FwdIter2>>::type uninitialized_copy_n(FwdIter1 first1, Size count, FwdIter2 dest, Sent2 last2);
```

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size**: The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2**: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **InIter2**.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.

- *first1*: Refers to the beginning of the sequence of elements that will be copied from.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *first2*: Refers to the beginning of the destination range.
- *last1*: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_copy_n* algorithm returns a *hpx::future<in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_copy_n* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

namespace hpx

namespace ranges

Functions

**template<typename FwdIter, typename Sent>
FwdIter uninitialized_default_construct (FwdIter first, Sent last)**

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

Return The *uninitialized_default_construct* algorithm returns a returns *FwdIter*. The *uninitialized_default_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

**template<typename ExPolicy, typename FwdIter, typename Sent>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_default_construct (ExPolicy
&&policy,
FwdIter
first,
Sent
last)**

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized stor-

age designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **FwdIter**.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_default_construct* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_default_construct* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename Rng>
hpx::traits::range_traits<Rng>::iterator_type uninitialized_default_construct (Rng
&&rng)
Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.
```

The assignments in the parallel *uninitialized_default_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **Rng**: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- **rng**: Refers to the range to which will be default constructed.

Return The *uninitialized_default_construct* algorithm returns a *hpx::traits::range_traits<Rng> ::iterator_type*. The *uninitialized_default_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

```
template<typename ExPolicy, typename Rng>
```

```
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type unininit
```

Constructs objects of type `typename iterator_traits<ForwardIt> ::value_type` in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the range to which the value will be default constructed

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_default_construct* algorithm returns a `hpx::future<typename hpx::traits::range_traits<Rng> ::iterator_type>`, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `typename hpx::traits::range_traits<Rng>::iterator_type` otherwise. The *uninitialized_default_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

```
template<typename FwdIter, typename Size>
```

```
FwdIter uninitialized_default_construct_n(FwdIter first, Size count)
```

Constructs objects of type `typename iterator_traits<ForwardIt> ::value_type` in the uninitialized storage designated by the range $[first, first + count]$ by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct_n* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count > 0*, no assignments otherwise.

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Size`: The type of the argument specifying the number of elements to apply *f* to.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at *first* the algorithm will be applied to.

Return The *uninitialized_default_construct_n* algorithm returns a returns *FwdIter*. The *uninitialized_default_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename ExPolicy, typename FwdIter, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_default_construct_n(ExPolicy  

&&policy,  

FwdIter  

first,  

Size  

count)
```

Constructs objects of type `typename iterator_traits<ForwardIt> ::value_type` in the uninitialized storage designated by the range `[first, first + count]` by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel `uninitialized_default_construct_n` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly `count` assignments, if `count > 0`, no assignments otherwise.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Size`: The type of the argument specifying the number of elements to apply `f` to.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at `first` the algorithm will be applied to.

The assignments in the parallel `uninitialized_default_construct_n` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `uninitialized_default_construct_n` algorithm returns a `hpx::future<FwdIter>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. The `uninitialized_default_construct_n` algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter, typename Sent, typename T>
FwdIter uninitialized_fill(FwdIter first, Sent last, T const &value)
```

Copies the given `value` to an uninitialized memory area, defined by the range `[first, last)`. If an exception is thrown during the initialization, the function has no effects.

The assignments in the ranges `uninitialized_fill` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between `first` and `last`

Template Parameters

- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `T`: The type of the value to be assigned (deduced).

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `value`: The value to be assigned.

Return The `uninitialized_fill` algorithm returns a returns `FwdIter`. The `uninitialized_fill` algorithm returns the output iterator to the element in the range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter, typename Sent>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_fill(ExPolicy
&&policy,
FwdIter
first,
Sent
last, T
const
&value)
```

Copies the given `value` to an uninitialized memory area, defined by the range [`first`, `last`). If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel `uninitialized_fill` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between `first` and `last`

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.
- `T`: The type of the value to be assigned (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- `value`: The value to be assigned.

The assignments in the parallel `uninitialized_fill` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `uninitialized_fill` algorithm returns a returns `FwdIter`. The `uninitialized_fill` algorithm returns the output iterator to the element in the range, one past the last element copied.

```
template<typename Rng, typename T>
```

`hpx::traits::range_traits<Rng>::iterator_type uninitialized_fill(Rng &&rng, T const &value)`

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *T*: The type of the value to be assigned (deduced).

Parameters

- *rng*: Refers to the range to which the value will be filled
- *value*: The value to be assigned.

Return The *uninitialized_fill* algorithm returns a returns `hpx::traits::range_traits<Rng> ::iterator_type`. The *uninitialized_fill* algorithm returns the output iterator to the element in the range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **T**>
`parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type uninitialized_fill(Rng &&rng, T const &value)`

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Linear in the distance between *first* and *last*

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- *T*: The type of the value to be assigned (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the range to which the value will be filled
- *value*: The value to be assigned.

The assignments in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_fill* algorithm returns a `hpx::future<typename hpx::traits::range_traits<Rng> ::iterator_type>`, if the execution policy is of

type *sequenced_task_policy* or *parallel_task_policy* and returns *typename hpx::traits::range_traits<Rng>::iterator_type* otherwise. The *uninitialized_fill* algorithm returns the iterator to one past the last element filled in the range.

```
template<typename FwdIter, typename Size, typename T>
FwdIter uninitialized_fill_n(FwdIter first, Size count, T const &value)
```

Copies the given *value* value to the first *count* elements in an uninitialized memory area beginning at *first*. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count* > 0, no assignments otherwise.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *T*: The type of the value to be assigned (deduced).

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *value*: The value to be assigned.

Return The *uninitialized_fill_n* algorithm returns a returns *FwdIter*. The *uninitialized_fill_n* algorithm returns the output iterator to the element in the range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename T>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_fill_n(ExPolicy
&&policy,
FwdIter
first,
Size
count,
T
const
&value)
```

Copies the given *value* value to the first *count* elements in an uninitialized memory area beginning at *first*. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count* > 0, no assignments otherwise.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *T*: The type of the value to be assigned (deduced).

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `count`: Refers to the number of elements starting at *first* the algorithm will be applied to.
- `value`: The value to be assigned.

The assignments in the parallel `uninitialized_fill_n` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The `uninitialized_fill_n` algorithm returns a `hpx::future<FwdIter>`, if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `FwdIter` otherwise. The `uninitialized_fill_n` algorithm returns the output iterator to the element in the range, one past the last element copied.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename InIter, typename Sent1, typename FwdIter, typename Sent2>
hpx::parallel::util::in_out_result<InIter, FwdIter> uninitialized_move(InIter first1, Sent1
last1, FwdIter first2,
Sent2 last2)
```

Moves the elements in the range, defined by `[first, last)`, to an uninitialized memory area beginning at `dest`. If an exception is thrown during the initialization, some objects in `[first, last)` are left in a valid but unspecified state.

The assignments in the parallel `uninitialized_move` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly `last - first` assignments.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `Sent1`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter`.
- `FwdIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- `Sent2`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `InIter2`.

Parameters

- `first1`: Refers to the beginning of the sequence of elements that will be moved from
- `last1`: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied
- `first2`: Refers to the beginning of the destination range.
- `last2`: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Return The `uninitialized_move` algorithm returns an `in_out_result<InIter, FwdIter>`. The `uninitialized_move` algorithm returns an input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2>
```

```
parallel::util::detail::algorithm_result<ExPolicy, parallel::util::in_out_result<FwdIter1, FwdIter2>>::type uninitialized_
```

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Sent1*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter*.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- *Sent2*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter2*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements that will be moved from
- *last1*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *first2*: Refers to the beginning of the destination range.
- *last2*: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

The assignments in the parallel *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<in_out_result<InIter, FwdIter>>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<InIter, FwdIter>* otherwise. The *uninitialized_move* algorithm returns an input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

```
template<typename Rng1, typename Rng2>
hpx::parallel::util::in_out_result<typename hpx::traits::range_traits<Rng1>::iterator_type, typename hpx::traits::range_
```

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- Rng1: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- Rng2: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- rng1: Refers to the range from which the elements will be moved from
- rng2: Refers to the range to which the elements will be moved to

Return The *uninitialized_move* algorithm returns an *in_out_result<typename hpx::traits::range_traits<Rng1> ::iterator_type, typename hpx::traits::range_traits<Rng2> ::iterator_type>*. The *uninitialized_move* algorithm returns an input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

```
template<typename ExPolicy, typename Rng1, typename Rng2>
```

```
parallel::util::detail::algorithm_result<ExPolicy, hpx::parallel::util::in_out_result<typename hpx::traits::range_traits<Rng1> ::iterator_type, typename hpx::traits::range_traits<Rng2> ::iterator_type>>
```

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- ExPolicy: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- Rng1: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- Rng2: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- policy: The execution policy to use for the scheduling of the iterations.
- rng1: Refers to the range from which the elements will be moved from
- rng2: Refers to the range to which the elements will be moved to

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<in_out_result<InIter, FwdIter>>`, if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `in_out_result< typename hpx::traits::range_traits<Rng1>::iterator_type , typename hpx::traits::range_traits<Rng2>::iterator_type >` otherwise. The *uninitialized_move* algorithm returns the input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

```
template<typename InIter, typename Size, typename FwdIter, typename Sent2>
hpx::parallel::util::in_out_result<InIter, FwdIter> uninitialized_move_n(InIter first1, Size
                                                               count, FwdIter
                                                               first2, Sent2 last2)
```

Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* movements, if *count* > 0, no move operations otherwise.

Template Parameters

- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- `Size`: The type of the argument specifying the number of elements to apply *f* to.
- `FwdIter`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- `Sent2`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.

Parameters

- `first1`: Refers to the beginning of the sequence of elements that will be moved from
- `count`: Refers to the number of elements starting at *first* the algorithm will be applied to.
- `first2`: Refers to the beginning of the destination range.
- `last2`: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Return The *uninitialized_move_n* algorithm returns `in_out_result<InIter, FwdIter>`. The *uninitialized_move_n* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

```
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2, typename Sent2>
parallel::util::detail::algorithm_result<ExPolicy, parallel::util::in_out_result<FwdIter1, FwdIter2>>::type uninitialized
```

Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* movements, if *count* > 0, no move operations otherwise.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter1*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.
- *FwdIter2*: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- *Sent2*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *InIter2*.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first1*: Refers to the beginning of the sequence of elements that will be moved from
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.
- *first2*: Refers to the beginning of the destination range.
- *last1*: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_move_n* algorithm returns a *hpx::future<in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_move_n* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

namespace hpx

namespace ranges

Functions

```
template<typename FwdIter, typename Sent>
FwdIter uninitialized_value_construct(FwdIter first, Sent last)
```

Constructs objects of type *typename iterator_traits<ForwardIt> ::value_type* in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

Return The *uninitialized_value_construct* algorithm returns a *FwdIter*. The *uninitialized_value_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

```
template<typename ExPolicy, typename FwdIter, typename Sent>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_value_construct(ExPolicy
&&policy,
FwdIter
first,
Sent
last)
```

Constructs objects of type `typename iterator_traits<ForwardIt> ::value_type` in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy**: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.

Parameters

- **policy**: The execution policy to use for the scheduling of the iterations.
- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_value_construct* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_value_construct* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename Rng>
hpx::traits::range_traits<Rng>::iterator_type uninitialized_value_construct(Rng
&&rng)
```

Constructs objects of type `typename iterator_traits<ForwardIt> ::value_type` in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- *rng*: Refers to the range to which will be value constructed.

Return The *uninitialized_value_construct* algorithm returns a *hpx::traits::range_traits<Rng>::iterator_type*. The *uninitialized_value_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

```
template<typename ExPolicy, typename Rng>
```

```
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type uninitialized_value_construct(
```

Constructs objects of type *typename iterator_traits<ForwardIt>::value_type* in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs exactly *last - first* assignments.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *rng*: Refers to the range to which the value will be value constructed

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *uninitialized_value_construct* algorithm returns a *hpx::future<typename hpx::traits::range_traits<Rng>::iterator_type>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *typename hpx::traits::range_traits<Rng>::iterator_type* otherwise. The *uninitialized_value_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

```
template<typename FwdIter, typename Size>
```

```
FwdIter uninitialized_value_construct_n(FwdIter first, Size count)
```

Constructs objects of type *typename iterator_traits<ForwardIt>::value_type* in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct_n* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs exactly *count* assignments, if *count* > 0, no assignments otherwise.

Template Parameters

- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Size*: The type of the argument specifying the number of elements to apply *f* to.

Parameters

- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *count*: Refers to the number of elements starting at *first* the algorithm will be applied to.

Return The *uninitialized_value_construct_n* algorithm returns a returns *FwdIter*. The *uninitialized_value_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename ExPolicy, typename FwdIter, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_value_construct_n(ExPolicy
    &&policy,
    FwdIter
    first,
    Size
    count)
```

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct_n* algorithm invoked with an execution policy object of type *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.

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename FwdIter, typename Sent, typename Pred, typename Proj>
subrange_t<FwdIter, Sent> unique (FwdIter first, Sent last, Pred &&pred, Proj &&proj)
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [*first*, *last*) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first - 1* applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- **FwdIter**: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent**: The type of the source sentinel (deduced). This sentinel type must be a sentinel for **FwdIter**.
- **Pred**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj**: The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [*first*, *last*). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

- **proj**: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *unique* algorithm returns `subrange_t<FwdIter, Sent>`. The *unique* algorithm returns an object {*ret*, *last*}, where *ret* is a past-the-end iterator for a new subrange.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Pred, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter, Sent>>::type unique (ExPolicy
    &&pol-
    icy,
    FwdIter
    first,
    Sent
    last,
    Pred
    &&pred,
    Proj
    &&proj)
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [*first*, *last*) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than *last - first* assignments, exactly *last - first - 1* applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- *ExPolicy*: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- *FwdIter*: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- *Sent*: The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *policy*: The execution policy to use for the scheduling of the iterations.
- *first*: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- *last*: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *unique* algorithm returns *subrange_t<FwdIter, Sent>*. The *unique* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange.

```
template<typename Rng, typename Pred, typename Proj>
subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> unique(Rng
    &&rng,
    Pred
    &&pred,
    Proj
    &&proj)
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range *rng* and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- *Rng*: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- *Pred*: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- *Proj*: The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- *rng*: Refers to the sequence of elements the algorithm will be applied to.
- *pred*: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- *proj*: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Return The *unique* algorithm returns *subrange_t<typename hpx::traits::range_iterator<Rng>::type,hpx::traits::range_iterator_t<Rng>>*. The *unique* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange.

```
namespace hpx
```

```
namespace parallel
```

```
namespace util
```

```
template<typename T = detail::no_data, typename Pred = std::less_equal<T>>
class cancellation_token
```

Public Functions

```
cancellation_token(T data)
bool was_cancelled(T data) const
void cancel(T data)
T get_data() const
```

Private Types

```
typedef std::atomic<T> flag_type
```

Private Members

```
std::shared_ptr<flag_type> was_cancelled_
```

```
template<typename Compare>
struct compare_projected<Compare, util::projection_identity>
```

Public Functions

```
template<typename Compare_>
constexpr compare_projected(Compare_ &&comp, util::projection_identity)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

```
Compare comp_
```

```
template<typename Compare, typename Proj2>
struct compare_projected<Compare, util::projection_identity, Proj2>
```

Public Functions

```
template<typename Compare_, typename Proj2_>
constexpr compare_projected(Compare_ &&comp, util::projection_identity, Proj2_ &&proj2)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

```
Compare comp_
```

```
Proj2 proj2_
```

```
template<typename Compare, typename Proj1>
struct compare_projected<Compare, Proj1, util::projection_identity>
```

Public Functions

```
template<typename Compare_, typename Proj1_>
constexpr compare_projected(Compare_ &&comp, Proj1_ &&proj1, util::projection_identity)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

```
Compare comp_
```

```
Proj1 proj1_
```

```
template<typename Compare>
struct compare_projected<Compare, util::projection_identity, util::projection_identity>
```

Public Functions

```
template<typename Compare_>
constexpr compare_projected(Compare_ &&comp, util::projection_identity,
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

```
Compare comp_
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace util
```

```
template<typename Compare, typename Proj>
struct compare_projected<Compare, Proj>
```

Public Functions

```
template<typename Compare_, typename Proj_>
constexpr compare_projected(Compare_ &&comp, Proj_ &&proj)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

Compare `comp_`

Proj `proj_`

```
template<typename Compare, typename Proj1, typename Proj2>
struct compare_projected<Compare, Proj1, Proj2>
```

Public Functions

```
template<typename Compare_, typename Proj1_, typename Proj2_>
constexpr compare_projected(Compare_ &&comp, Proj1_ &&proj1, Proj2_
&&proj2)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

Compare `comp_`

Proj1 `proj1_`

Proj2 `proj2_`

```
template<typename Compare, typename Proj1>
struct compare_projected<Compare, Proj1, util::projection_identity>
```

Public Functions

```
template<typename Compare_, typename Proj1_>
constexpr compare_projected(Compare_ &&comp, Proj1_ &&proj1,
util::projection_identity)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

Compare `comp_`

Proj1 `proj1_`

```
template<typename Compare>
struct compare_projected<Compare, util::projection_identity>
```

Public Functions

```
template<typename Compare_>
constexpr compare_projected(Compare_ &&comp, util::projection_identity)

template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

Compare `comp_`

```
template<typename Compare, typename Proj2>
struct compare_projected<Compare, util::projection_identity, Proj2>
```

Public Functions

```
template<typename Compare_, typename Proj2_>
constexpr compare_projected(Compare_ &&comp, util::projection_identity, Proj2_
&&proj2)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

Compare `comp_`

Proj2 `proj2_`

```
template<typename Compare>
struct compare_projected<Compare, util::projection_identity, util::projection_identity>
```

Public Functions

```
template<typename Compare_>
constexpr compare_projected(Compare_ &&comp, util::projection_identity,
util::projection_identity)
```

```
template<typename T1, typename T2>
constexpr bool operator()(T1 &&t1, T2 &&t2) const
```

Public Members

Compare `comp_`

```
namespace hpx
```

```
namespace parallel
```

```
namespace util

template<typename Pred, typename Proj>
struct invoke_projected
```

Public Types

```
template<>
using pred_type = typename std::decay<Pred>::type
template<>
using proj_type = typename std::decay<Proj>::type
```

Public Functions

```
template<typename Pred_, typename Proj_>
invoke_projected(Pred_ &&pred, Proj_ &&proj)

template<typename T>
decltype(auto) operator()(T &&t)

template<typename T>
decltype(auto) operator()(T &&t, T &&u)
```

Public Members

```
pred_type pred_
proj_type proj_

template<typename Pred>
struct invoke_projected<Pred, projection_identity>
```

Public Types

```
template<>
using pred_type = typename std::decay<Pred>::type
```

Public Functions

```
template<typename Pred_>
invoke_projected(Pred_ &&pred, projection_identity)

template<typename T>
decltype(auto) operator()(T &&t)

template<typename T>
auto operator()(T &&t, T &&u)
```

Public Members

```
pred_type pred_
namespace hpx
namespace parallel
namespace util
```

Functions

```
template<typename Iter, typename F, typename Cleanup>
constexpr Iter loop_with_cleanup(Iter it, Iter last, F &&f, Cleanup &&cleanup)

template<typename Iter, typename FwdIter, typename F, typename Cleanup>
constexpr FwdIter loop_with_cleanup(Iter it, Iter last, FwdIter dest, F &&f, Cleanup
&&cleanup)

template<typename Iter, typename F, typename Cleanup>
constexpr Iter loop_with_cleanup_n(Iter it, std::size_t count, F &&f, Cleanup
&&cleanup)

template<typename Iter, typename FwdIter, typename F, typename Cleanup>
constexpr FwdIter loop_with_cleanup_n(Iter it, std::size_t count, FwdIter dest, F
&&f, Cleanup &&cleanup)

template<typename Iter, typename CancelToken, typename F, typename Cleanup>
constexpr Iter loop_with_cleanup_n_with_token(Iter it, std::size_t count, CancelToken &tok, F &&f, Cleanup
&&cleanup)

template<typename Iter, typename FwdIter, typename CancelToken, typename F, typename Cleanup>
constexpr FwdIter loop_with_cleanup_n_with_token(Iter it, std::size_t count,
FwdIter dest, CancelToken
&tok, F &&f, Cleanup
&&cleanup)

template<typename Iter, typename T, typename Pred>
T accumulate_n(Iter it, std::size_t count, T init, Pred &&f)

template<typename T, typename Iter, typename Reduce, typename Conv = util::projection_identity>
T accumulate(Iter first, Iter last, Reduce &&r, Conv &&conv = Conv())

template<typename T, typename Iter1, typename Iter2, typename Reduce, typename Conv>
T accumulate(Iter1 first1, Iter1 last1, Iter2 first2, Reduce &&r, Conv &&conv)
```

Variables

```
template<typename ExPolicy>
constexpr loop_step_t<ExPolicy> loop_step = loop_step_t<ExPolicy>{ }

template<typename ExPolicy>
constexpr loop_optimization_t<ExPolicy> loop_optimization = loop_optimization_t<ExPolicy>{ }

constexpr loop_t loop = loop_t{ }

constexpr loop_ind_t loop_ind = loop_ind_t{ }

template<typename ExPolicy>
constexpr loop2_t<ExPolicy> loop2 = loop2_t<ExPolicy>{ }

template<typename ExPolicy>
constexpr loop_n_t<ExPolicy> loop_n = loop_n_t<ExPolicy>{ }

template<typename ExPolicy>
constexpr loop_n_ind_t<ExPolicy> loop_n_ind = loop_n_ind_t<ExPolicy>{ }

template<typename ExPolicy>
constexpr loop_idx_n_t<ExPolicy> loop_idx_n = loop_idx_n_t<ExPolicy>{ }

template<typename ExPolicy>
struct loop2_t : public hpx::functional::detail::tag_fallback<loop2_t<ExPolicy>>
```

Friends

```
template<typename VecOnly, typename Begin1, typename End1, typename Begin2, typename F>
friend constexpr std::pair<Begin1, Begin2> tag_invoke (hpx::parallel::util::loop2_t<ExPolicy>,
VecOnly&&,
Begin1 begin1,
End1 end1,
Begin2 begin2,
F &&f)
```



```
template<typename ExPolicy>
struct loop_idx_n_t : public hpx::functional::detail::tag_fallback<loop_idx_n_t<ExPolicy>>
```

Friends

```
template<typename Iter, typename F>
friend constexpr Iter tag_invoke (hpx::parallel::util::loop_idx_n_t<ExPolicy>,
std::size_t base_idx, Iter it,
std::size_t count, F &&f)
```



```
template<typename Iter, typename CancelToken, typename F>
friend constexpr Iter tag_invoke (hpx::parallel::util::loop_idx_n_t<ExPolicy>,
std::size_t base_idx, Iter it,
std::size_t count, CancelToken
&tok, F &&f)
```



```
struct loop_ind_t : public hpx::functional::detail::tag_fallback<loop_ind_t>
```

Friends

```
template<typename ExPolicy, typename Begin, typename End, typename F>
friend constexpr Begin tag_fallback_invoke (hpx::parallel::util::loop_ind_t,
                                            ExPolicy&&, Begin begin, End end, F &&f)
```

```
template<typename ExPolicy, typename Begin, typename End, typename CancelToken, typename F>
friend constexpr Begin tag_fallback_invoke (hpx::parallel::util::loop_ind_t,
                                            ExPolicy&&, Begin begin, End end, CancelToken &tok, F &&f)
```

```
template<typename ExPolicy>
struct loop_n_ind_t : public hpx::functional::detail::tag_fallback<loop_n_ind_t<ExPolicy>>
```

Friends

```
template<typename Iter, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_ind_t<ExPolicy>,
                                            Iter it, std::size_t count, F &&f)
```

```
template<typename Iter, typename CancelToken, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_ind_t<ExPolicy>,
                                            Iter it, std::size_t count, CancelToken &tok, F &&f)
```

```
template<typename ExPolicy>
struct loop_n_t : public hpx::functional::detail::tag_fallback<loop_n_t<ExPolicy>>
```

Friends

```
template<typename Iter, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_t<ExPolicy>,
                                            Iter it, std::size_t count, F &&f)
```

```
template<typename Iter, typename CancelToken, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_t<ExPolicy>,
                                            Iter it, std::size_t count, CancelToken &tok, F &&f)
```

```
template<typename ExPolicy>
struct loop_optimization_t : public hpx::functional::detail::tag_fallback<loop_optimization_t<ExPolicy>>
```

Friends

```
template<typename Iter>
friend constexpr bool tag_fallback_invoke (hpx::parallel::util::loop_optimization_t<ExPolicy>,
                                             Iter, Iter)
```

```
template<typename ExPolicy>
struct loop_step_t : public hpx::functional::detail::tag_fallback<loop_step_t<ExPolicy>>
```

Friends

```
template<typename VecOnly, typename F, typename ...Iters>
hpx::util::invoke_result<F, Iters...>::type tagFallbackInvoke (hpx::parallel::util::loop_step_t<ExPolicyVecOnly&&, F &&f,
    Iters&... its)

struct loop_t : public hpx::functional::detail::tagFallback<loop_t>
```

Friends

```
template<typename ExPolicy, typename Begin, typename End, typename F>
friend constexpr Begin tagFallbackInvoke (hpx::parallel::util::loop_t, ExPolicy&&, Begin begin, End end, F
    &&f)

template<typename ExPolicy, typename Begin, typename End, typename CancelToken, typename F>
friend constexpr Begin tagFallbackInvoke (hpx::parallel::util::loop_t, ExPolicy&&, Begin begin, End end,
    CancelToken &tok, F &&f)
```

namespace hpx

```
namespace parallel
```

namespace util

Functions

```
template<typename Value, typename ...Args>
void construct_object (Value *ptr, Args&&... args)
    create an object in the memory specified by ptr
```

Template Parameters

- **Value**: : typename of the object to create
- **Args**: : parameters for the constructor

Parameters

- [in] **ptr**: : pointer to the memory where to create the object
- [in] **args**: : arguments to the constructor

```
template<typename Value>
void destroy_object (Value *ptr)
    destroy an object in the memory specified by ptr
```

Template Parameters

- **Value**: : typename of the object to create

Parameters

- [in] **ptr**: : pointer to the object to destroy

```
template<typename Iter, typename Sent>
void init (Iter first, Sent last, typename std::iterator_traits<Iter>::value_type &val)
    Initialize a range of objects with the object val moving across them
```

Return range initialized

Parameters

- [in] r: : range of elements not initialized
- [in] val: : object used for the initialization

template<typename **Value**, typename ...**Args**>

void **construct** (*Value* *ptr, *Args*&&... args)

create an object in the memory specified by ptr

Template Parameters

- Value: : typename of the object to create
- Args: : parameters for the constructor

Parameters

- [in] ptr: : pointer to the memory where to create the object
- [in] args: : arguments to the constructor

template<typename **Iter1**, typename **Sent1**, typename **Iter2**>

Iter2 **init_move** (*Iter2* it_dest, *Iter1* first, *Sent1* last)

Move objects.

Template Parameters

- Iter: : iterator to the elements
- Value: : typename of the object to create

Parameters

- [in] itdest: : iterator to the final place of the objects
- [in] R: : range to move

template<typename **Iter**, typename **Sent**, typename **Value** = **typename** std::iterator_traits<*Iter*>::value_type>

Value ***uninit_move** (*Value* *ptr, *Iter* first, *Sent* last)

Move objects to uninitialized memory.

Template Parameters

- Iter: : iterator to the elements
- Value: : typename of the object to construct

Parameters

- [in] ptr: : pointer to the memory where to create the object
- [in] R: : range to move

template<typename **Iter**, typename **Sent**>

void **destroy** (*Iter* first, *Sent* last)

Move objects to uninitialized memory.

Template Parameters

- Iter: : iterator to the elements
- Value: : typename of the object to construct

Parameters

- [in] ptr: : pointer to the memory where to construct the object
- [in] R: : range to move

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Compare**>

Iter2 **full_merge** (*Iter1* buf1, *Sent1* end_buf1, *Iter1* buf2, *Sent1* end_buf2, *Iter2* buf_out, *Compare* comp)

Merge two contiguous buffers pointed by buf1 and buf2 , and put in the buffer pointed by buf_out.

Parameters

- [in] buf1: : iterator to the first element in the first buffer
- [in] end_buf1: : final iterator of first buffer
- [in] buf2: : iterator to the first iterator to the second buffer
- [in] end_buf2: : final iterator of the second buffer
- [in] buf_out: : buffer where move the elements merged
- [in] comp: : comparison object

```
template<typename Iter, typename Sent, typename Value, typename Compare>
Value *uninit_full_merge(Iter first1, Sent last1, Iter first2, Sent last2, Value *it_out, Compare comp)
```

Merge two contiguous buffers pointed by first1 and first2 , and put in the uninitialized buffer pointed by it_out.

Parameters

- [in] first1: : iterator to the first element in the first buffer
- [in] last: : last iterator of the first buffer
- [in] first2: : iterator to the first element to the second buffer
- [in] last2: : final iterator of the second buffer
- [in] it_out: : uninitialized buffer where move the elements merged
- [in] comp: : comparison object

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
Iter2 half_merge(Iter1 buf1, Sent1 end_buf1, Iter2 buf2, Sent2 end_buf2, Iter2 buf_out, Compare comp)
```

: Merge two buffers. The first buffer is in a separate memory. The second buffer have a empty space before buf2 of the same size than the (end_buf1 - buf1)

Remark The elements pointed by Iter1 and Iter2 must be the same

Parameters

- [in] buf1: : iterator to the first element of the first buffer
- [in] end_buf1: : iterator to the last element of the first buffer
- [in] buf2: : iterator to the first element of the second buffer
- [in] end_buf2: : iterator to the last element of the second buffer
- [in] buf_out: : iterator to the first element to the buffer where put the result
- [in] comp: : object for Compare two elements of the type pointed by the Iter1 and Iter2

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Compare>
bool in_place_merge_uncontiguous(Iter1 src1, Sent1 end_src1, Iter2 src2, Sent2 end_src2, Iter3 aux, Compare comp)
```

Merge two non contiguous buffers, placing the results in the buffers for to do this use an auxiliary buffer pointed by aux

Parameters

- [in] src1: : iterator to the first element of the first buffer
- [in] end_src1: : last iterator of the first buffer
- [in] src2: : iterator to the first element of the second buffer
- [in] end_src2: : last iterator of the second buffer
- [in] aux: : iterator to the first element of the auxiliary buffer
- [in] comp: : object for to Compare elements

Exceptions

-

```
template<typename Iter1, typename Sent1, typename Iter2, typename Compare>
```

```
bool in_place_merge(Iter1 src1, Iter1 src2, Sent1 end_src2, Iter2 buf, Compare comp)
: merge two contiguous buffers,using an auxiliary buffer pointed by buf
```

Parameters

- [in] src1: iterator to the first position of the first buffer
- [in] src2: final iterator of the first buffer and first iterator of the second buffer
- [in] end_src2: final iterator of the second buffer
- [in] buf: iterator to buffer used as auxiliary memory
- [in] comp: object for to Compare elements

Exceptions

-

```
namespace hpx
```

```
namespace parallel
```

```
namespace util
```

Functions

```
template<typename Iter, typename Sent, typename Compare>
bool less_range(Iter it1, std::uint32_t pos1, Sent it2, std::uint32_t pos2, Compare comp)
Compare the elements pointed by it1 and it2, and if they are equals, compare their position, doing
a stable comparison.
```

Return result of the comparison**Parameters**

- [in] it1: iterator to the first element
- [in] pos1: position of the object pointed by it1
- [in] it2: iterator to the second element
- [in] pos2: position of the element pointed by it2
- [in] comp: comparison object

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
util::range<Iter1, Sent1> full_merge4(util::range<Iter1, Sent1> &rdest, util::range<Iter2,
                                         Sent2> vrangle_input[4], std::uint32_t nrange_input,
                                         Compare comp)
```

Merge four ranges.

Return range with all the elements move with the size adjusted**Parameters**

- [in] dest: range where move the elements merged. Their size must be greater or equal than the sum of the sizes of the ranges in the array R
- [in] R: array of ranges to merge
- [in] nrange_input: number of ranges in R
- [in] comp: comparison object

```
template<typename Value, typename Iter, typename Sent, typename Compare>
util::range<Value*> uninit_full_merge4(util::range<Value*> const &dest,
                                         util::range<Iter, Sent> vrangle_input[4],
                                         std::uint32_t nrange_input, Compare comp)
```

Merge four ranges and put the result in uninitialized memory.

Return range with all the elements move with the size adjusted

Parameters

- [in] dest: range where create and move the elements merged. Their size must be greater or equal than the sum of the sizes of the ranges in the array R
- [in] R: : array of ranges to merge
- [in] nrange_input: : number of ranges in vrange_input
- [in] comp: : comparison object

namespace hpx

namespace parallel

namespace util

Functions

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
void merge_level4(util::range<Iter1, Sent1> dest, std::vector<util::range<Iter2, Sent2>>
&v_input, std::vector<util::range<Iter1, Sent1>> &v_output, Compare
comp)
```

Merge the ranges in the vector v_input using full_merge4. The v_output vector is used as auxiliary memory in the internal process. The final results is in the dest range. All the ranges of v_output are inside the range dest

Return range with all the elements moved

Parameters

- [in] dest: : range where move the elements merged
- [in] v_input: : vector of ranges to merge
- [in] v_output: : vector of ranges obtained
- [in] comp: : comparison object

```
template<typename Value, typename Iter, typename Sent, typename Compare>
void uninit_merge_level4(util::range<Value*> dest, std::vector<util::range<Iter, Sent>>
&v_input, std::vector<util::range<Value*>> &v_output,
Compare comp)
```

Merge the ranges over uninitialized memory,in the vector v_input using full_merge4. The v_output vector is used as auxiliary memory in the internal process. The final results is in the dest range. All the ranges of v_output are inside the range dest

Return range with all the elements moved

Parameters

- [in] dest: : range where move the elements merged
- [in] v_input: : vector of ranges to merge
- [in] v_output: : vector of ranges obtained
- [in] comp: : comparison object

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
util::range<Iter2, Sent2> merge_vector4(util::range<Iter1, Sent1> range_input,
util::range<Iter2, Sent2> range_output,
std::vector<util::range<Iter1, Sent1>> &v_input,
std::vector<util::range<Iter2, Sent2>> &v_output,
Compare comp)
```

Merge the ranges in the vector v_input using merge_level4. The v_output vector is used as auxiliary memory in the internal process. The final results is in the range_output range. All the ranges

of `v_output` are inside the range `range_output`. All the ranges of `v_input` are inside the range `range_input`.

Parameters

- [in] range_input: : range including all the ranges of v_input
 -

namespace hpx

namespace parallel

namespace util

Functions

constexpr *std::uint32_t nbits32 (std::uint32_t num)*

Obtain the number of bits equal or greater than num.

Return Number of bits

Parameters

- [in] num: : Number to examine

Exceptions

- none:

constexpr *std::uint32_t nbits64* (*std::uint64_t num*)

Obtain the number of bits equal or greater than num.

Return Number of bits

Parameters

- [in] num: : Number to examine

Exceptions

- none:

Variables

```
namespace hpx
```

namespace parallel

namespace util

Functions

```
template<typename Itr, typename ...Ts>
prefetching::prefetcher_context<Itr, Ts const...> make_prefetcher_context (Itr
    base_begin,
    Itr
    base_end,
    std::size_t
    p_factor,
    Ts
    const&...
    rngs)
```

namespace prefetching

Functions

```
template<typename ...Ts, std::size_t... Is>
void prefetch_containers (hpx::tuple<Ts...> const &t, hpx::util::index_pack<Is...>,
    std::size_t idx)

template<typename ExPolicy, typename Itr, typename ...Ts, typename F>
constexpr prefetching_iterator<Itr, Ts...> tag_invoke (hpx::parallel::util::loop_n_t<ExPolicy>,
    prefetching_iterator<Itr, Ts...>
    it, std::size_t count, F &&f)

template<typename ExPolicy, typename Itr, typename ...Ts, typename F>
constexpr prefetching_iterator<Itr, Ts...> tag_invoke (hpx::parallel::util::loop_n_ind_t<ExPolicy>,
    prefetching_iterator<Itr, Ts...>
    it, std::size_t count, F &&f)

struct loop_n_helper
```

Public Static Functions

```
template<typename Itr, typename ...Ts, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr,
    Ts...> it, std::size_t count, F
    &&f, Pred)

template<typename Itr, typename ...Ts, typename CancelToken, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr,
    Ts...> it, std::size_t count,
    CancelToken &tok, F &&f,
    Pred)

struct loop_n_ind_helper
```

Public Static Functions

```
template<typename Itr, typename ...Ts, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr,
    Ts...> it, std::size_t count, F
    &&f, Pred)

template<typename Itr, typename ...Ts, typename CancelToken, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr,
    Ts...> it, std::size_t count,
    CancelToken &tok, F &&f,
    Pred)

template<typename Itr, typename ...Ts>
struct prefetcher_context
```

Public Functions

```
prefetcher_context (Itr begin, Itr end, ranges_type const &rngs, std::size_t
    p_factor = 1)

prefetching_iterator<Itr, Ts...> begin ()

prefetching_iterator<Itr, Ts...> end ()
```

Private Types

```
typedef hpx::tuple<std::reference_wrapper<Ts>...> ranges_type
```

Private Members

```
Itr it_begin_
Itr it_end_
ranges_type rngs_
std::size_t chunk_size_
std::size_t range_size_
```

Private Static Attributes

```
constexpr std::size_t sizeof_first_value_type = sizeof(typename hpx::tuple_element<0, ranges_
```

```
template<typename Itr, typename ...Ts>
class prefetching_iterator
```

Public Types

```
template<>
using base_iterator = Itr

template<>
using iterator_category = std::random_access_iterator_tag

template<>
using value_type = typename std::iterator_traits::value_type

template<>
using difference_type = std::ptrdiff_t

template<>
using pointer = value_type*

template<>
using reference = value_type&
```

Public Functions

```
prefetching_iterator(std::size_t idx, base_iterator base, std::size_t chunk_size,
                      std::size_t range_size, ranges_type const &rngs)

ranges_type const &ranges() const

Itr base() const

std::size_t chunk_size() const

std::size_t range_size() const

std::size_t index() const

prefetching_iterator &operator+=(difference_type rhs)

prefetching_iterator &operator-=(difference_type rhs)

prefetching_iterator &operator++()

prefetching_iterator &operator--()

prefetching_iterator operator++(int)

prefetching_iterator operator--(int)

difference_type operator-(const prefetching_iterator &rhs) const

bool operator==(const prefetching_iterator &rhs) const

bool operator!=(const prefetching_iterator &rhs) const

bool operator>(const prefetching_iterator &rhs) const

bool operator<(const prefetching_iterator &rhs) const

bool operator>=(const prefetching_iterator &rhs) const

bool operator<=(const prefetching_iterator &rhs) const
```

```
    std::size_t &operator[] (std::size_t)
    std::size_t operator* () const
```

Private Types

```
template<>
using ranges_type = hpx::tuple<std::reference_wrapper<Ts>...>
```

Private Members

```
ranges_type rngs_
base_iterator base_
std::size_t chunk_size_
std::size_t range_size_
std::size_t idx_
```

Friends

```
prefetching_iterator operator+ (prefetching_iterator const &lhs, difference_type rhs)
prefetching_iterator operator- (prefetching_iterator const &lhs, difference_type rhs)

namespace hpx
```

Typedefs

```
using identity = hpx::parallel::util::projection_identity

namespace parallel

namespace util

struct projection_identity
```

Public Types

```
using is_transparent = std::true_type
```

Public Functions

```
template<typename T>
constexpr T &&operator() (T &&val) const

namespace hpx

namespace parallel

namespace util
```

Typedefs

```
template<typename Iterator, typename Sentinel = Iterator>
using range = hpx::util::iterator_range<Iterator, Sentinel>
```

Functions

```
template<typename Iter, typename Sent>
range<Iter, Sent> concat (range<Iter, Sent> const &it1, range<Iter, Sent> const &it2)
    concatenate two contiguous ranges
```

Return range resulting of the concatenation

Parameters

- [in] it1:: first range
- [in] it2:: second range

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2>
range<Iter2, Iter2> init_move (range<Iter2, Sent2> const &dest, range<Iter1, Sent1>
                                const &src)
Move objects from the range src to dest.
```

Return range with the objects moved and the size adjusted

Parameters

- [in] dest:: range where move the objects
- [in] src:: range from where move the objects

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2>
range<Iter2, Sent2> uninit_move (range<Iter2, Sent2> const &dest, range<Iter1, Sent1>
                                const &src)
Move objects from the range src creating them in dest.
```

Return range with the objects moved and the size adjusted

Parameters

- [in] dest:: range where move and create the objects
- [in] src:: range from where move the objects

```
template<typename Iter, typename Sent>
void destroy_range (range<Iter, Sent> r)
    destroy a range of objects
```

Parameters

- [in] r: : range to destroy

```
template<typename Iter, typename Sent>
range<Iter, Sent> init(range<Iter, Sent> const &r, typename
std::iterator_traits<Iter>::value_type &val)
initialize a range of objects with the object val moving across them
```

Return range initialized**Parameters**

- [in] r: : range of elements not initialized
- [in] val: : object used for the initialization

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
bool is_mergeable(range<Iter1, Sent1> const &src1, range<Iter2, Sent2> const &src2,
Compare comp)
: indicate if two ranges have a possible merge
```

Parameters

- [in] src1: : first range
- [in] src2: : second range
- [in] comp: : object for to compare elements

Exceptions

-

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Sent3>
range<Iter3, Sent3> full_merge(range<Iter3, Sent3> const &dest, range<Iter1, Sent1>
const &src1, range<Iter2, Sent2> const &src2, Compare comp)
```

Merge two contiguous ranges src1 and src2 , and put the result in the range dest, returning the range merged.

Return range with the elements merged and the size adjusted**Parameters**

- [in] dest: : range where locate the elements merged. the size of dest must be greater or equal than the sum of the sizes of src1 and src2
- [in] src1: : first range to merge
- [in] src2: : second range to merge
- [in] comp: : comparison object

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Value, typename Compare>
range<Value*> uninit_full_merge(const range<Value*> &dest, range<Iter1, Sent1>
const &src1, range<Iter2, Sent2> const &src2,
Compare comp)
```

Merge two contiguous ranges src1 and src2 , and create and move the result in the uninitialized range dest, returning the range merged.

Return range with the elements merged and the size adjusted**Parameters**

- [in] dest: : range where locate the elements merged. the size of dest must be greater or equal than the sum of the sizes of src1 and src2. Initially is uninitialized memory
- [in] src1: : first range to merge
- [in] src2: : second range to merge
- [in] comp: : comparison object

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
range<Iter2, Sent2> half_merge(range<Iter2, Sent2> const &dest, range<Iter1, Sent1>
const &src1, range<Iter2, Sent2> const &src2, Compare comp)
```

: Merge two buffers. The first buffer is in a separate memory

Return : range with the two buffers merged

Parameters

- [in] dest: : range where finish the two buffers merged
- [in] src1: : first range to merge in a separate memory
- [in] src2: : second range to merge, in the final part of the range where deposit the final results
- [in] comp: : object for compare two elements of the type pointed by the Iter1 and Iter2

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Sent3>
bool in_place_merge_uncontiguous(range<Iter1, Sent1> const &src1, range<Iter2, Sent2> const &src2, range<Iter3, Sent3> &aux,
                                Compare comp)
```

: merge two non contiguous buffers src1 , src2, using the range aux as auxiliary memory

Parameters

- [in] src1: : first range to merge
- [in] src2: : second range to merge
- [in] aux: : auxiliary range used in the merge
- [in] comp: : object for to compare elements

Exceptions

-

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
range<Iter1, Sent1> in_place_merge(range<Iter1, Sent1> const &src1, range<Iter1, Sent1> const &src2, range<Iter2, Sent2> &buf,
                                    Compare comp)
```

: merge two contiguous buffers (src1, src2) using buf as auxiliary memory

Parameters

- [in] src1: : first range to merge
- [in] src2: : second range to merge
- [in] buf: : auxiliary memory used in the merge
- [in] comp: : object for to compare elements

Exceptions

-

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
void merge_flow(range<Iter1, Sent1> rng1, range<Iter2, Sent2> rbuf, range<Iter1, Sent1>
                  rng2, Compare cmp)
```

: merge two contiguous buffers

Template Parameters

- Iter: : iterator to the elements
- compare: : object for to compare two elements pointed by Iter iterators

Parameters

- [in] first: : iterator to the first element
- [in] last: : iterator to the element after the last in the range

- [in] comp: : object for to compare elements

Exceptions

-

```
namespace hpx
```

```
namespace ranges
```

Functions

```
template<typename Iter>
constexpr Iter next (Iter first, typename std::iterator_traits<Iter>::difference_type dist = 1)

template<typename Iter, typename Sent>
constexpr Iter next (Iter first, Sent bound)

template<typename Iter, typename Sent>
constexpr Iter next_ (Iter first, typename std::iterator_traits<Iter>::difference_type n, Sent
bound)

template<typename Iter, typename Sent>
constexpr Iter next_ (Iter first, typename std::iterator_traits<Iter>::difference_type n, Sent
bound, std::true_type, std::true_type)

template<typename Iter, typename Sent>
constexpr Iter next_ (Iter first, typename std::iterator_traits<Iter>::difference_type n, Sent
bound)
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace util
```

Functions

```
template<typename I1, typename I2>
I2 get_in2_element (util::in_in_result<I1, I2> &&p)

template<typename I1, typename I2>
hpx::future<I2> get_in2_element (hpx::future<util::in_in_result<I1, I2>> &&f)

template<typename I, typename O>
std::pair<I, O> get_pair (util::in_out_result<I, O> &&p)

template<typename I, typename O>
O get_second_element (util::in_out_result<I, O> &&p)

template<typename I, typename O>
hpx::future<std::pair<I, O>> get_pair (hpx::future<util::in_out_result<I, O>> &&f)

template<typename I, typename O>
hpx::future<O> get_second_element (hpx::future<util::in_out_result<I, O>> &&f)
```

```
template<typename I, typename O>
hpx::util::iterator_range<I, O> get_subrange (in_out_result<I, O> const &iор)

template<typename I, typename O>
hpx::future<hpx::util::iterator_range<I, O>> get_subrange (hpx::future<in_out_result<I, O>> &&iор)

template<typename I1, typename I2, typename O>
O get_third_element (util::in_out_result<I1, I2, O> &&p)

template<typename I1, typename I2, typename O>
hpx::future<O> get_third_element (hpx::future<util::in_out_result<I1, I2, O>> &&f)

template<typename ...Ts>
constexpr in_out_out_result<Ts...> make_in_out_out_result (hpx::tuple<Ts...>
&&t)

template<typename ...Ts>
hpx::future<in_out_out_result<Ts...>> make_in_out_out_result (hpx::future<hpx::tuple<Ts...>>
&&f)

template<typename Iterator, typename Sentinel = Iterator>
hpx::util::iterator_range<Iterator, Sentinel> make_subrange (Iterator iterator, Sentinel sentinel)

template<typename Iterator, typename Sentinel = Iterator>
hpx::future<hpx::util::iterator_range<Iterator, Sentinel>> make_subrange (hpx::future<Iterator>
&&iterator, Sentinel sentinel)

template<typename I, typename F>
struct in_fun_result
```

Public Functions

```
template<typename I2, typename F2, typename Enable = std::enable_if_t<std::is_convertible_v<I const&, I2>>
constexpr operator in_fun_result<I2, F2> () const &

template<typename I2, typename F2, typename Enable = std::enable_if_t<std::is_convertible_v<I, I2> && std::is_convertible_v<F, F2>>
constexpr operator in_fun_result<I2, F2> () &&

template<typename Archive>
void serialize (Archive &ar, unsigned)
```

Public Members

```
HPX_NO_UNIQUE_ADDRESS I hpx::parallel::util::in_fun_result::in
HPX_NO_UNIQUE_ADDRESS F hpx::parallel::util::in_fun_result::fun

template<typename I1, typename I2, typename O>
struct in_in_out_result
```

Public Functions

```
template<typename II1, typename II2, typename O1, typename Enable = typename std::enable_if_t<std::is_
constexpr operator in_in_out_result<II1, II2, O1>() const &

template<typename II2, typename II1, typename O1, typename Enable = typename std::enable_if_t<std::is_
constexpr operator in_in_out_result<II1, II2, O1>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)
```

Public Members

```
HPX_NO_UNIQUE_ADDRESS I1 hpx::parallel::util::in_in_out_result::in1
HPX_NO_UNIQUE_ADDRESS I2 hpx::parallel::util::in_in_out_result::in2
HPX_NO_UNIQUE_ADDRESS O hpx::parallel::util::in_in_out_result::out

template<typename I1, typename I2>
struct in_in_result
```

Public Functions

```
template<typename II1, typename II2, typename Enable = std::enable_if_t<std::is_convertible_v<I1 const&
constexpr operator in_in_result<II1, II2>() const &

template<typename II1, typename II2, typename Enable = std::enable_if_t<std::is_convertible_v<I1, III> &&
constexpr operator in_in_result<II1, II2>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)
```

Public Members

```
HPX_NO_UNIQUE_ADDRESS I1 hpx::parallel::util::in_in_result::in1
HPX_NO_UNIQUE_ADDRESS I2 hpx::parallel::util::in_in_result::in2

template<typename I, typename O1, typename O2>
struct in_out_out_result
```

Public Functions

```
template<typename II, typename OO1, typename OO2, typename Enable = typename std::enable_if_t<std::is_
constexpr operator in_out_out_result<II, OO1, OO2>() const &

template<typename II, typename OO1, typename OO2, typename Enable = typename std::enable_if_t<std::is_
constexpr operator in_out_out_result<II, OO1, OO2>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)
```

Public Members

```
HPX_NO_UNIQUE_ADDRESS I hpx::parallel::util::in_out_out_result::in
HPX_NO_UNIQUE_ADDRESS O1 hpx::parallel::util::in_out_out_result::out1
HPX_NO_UNIQUE_ADDRESS O2 hpx::parallel::util::in_out_out_result::out2

template<typename I, typename O>
struct in_out_result
```

Public Functions

```
template<typename I2, typename O2, typename Enable = std::enable_if_t<std::is_convertible_v<I const&, I2>>
constexpr operator in_out_result<I2, O2>() const &

template<typename I2, typename O2, typename Enable = std::enable_if_t<std::is_convertible_v<I, I2> && std::is_convertible_v<O, O2>>
constexpr operator in_out_result<I2, O2>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)
```

Public Members

```
HPX_NO_UNIQUE_ADDRESS I hpx::parallel::util::in_out_result::in
HPX_NO_UNIQUE_ADDRESS O hpx::parallel::util::in_out_result::out

template<typename T>
struct min_max_result
```

Public Functions

```
template<typename T2, typename Enable = std::enable_if_t<std::is_convertible_v<T const&, T>>>
constexpr operator min_max_result<T2>() const &

template<typename T2, typename Enable = std::enable_if_t<std::is_convertible_v<T, T2>>>
constexpr operator min_max_result<T2>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)
```

Public Members

```
HPX_NO_UNIQUE_ADDRESS T hpx::parallel::util::min_max_result::min
HPX_NO_UNIQUE_ADDRESS T hpx::parallel::util::min_max_result::max
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace util
```

Functions

```
template<typename InIter, typename Sent, typename OutIter>
constexpr in_out_result<InIter, OutIter> copy (InIter first, Sent last, OutIter dest)

template<typename InIter, typename OutIter>
constexpr void copy_synchronize (InIter const &first, OutIter const &dest)

template<typename InIter, typename Sent, typename OutIter>
constexpr in_out_result<InIter, OutIter> move (InIter first, Sent last, OutIter dest)

template<typename InIter, typename OutIter>
constexpr in_out_result<InIter, OutIter> move_n (InIter first, std::size_t count, OutIter dest)
```

Variables

```
template<typename ExPolicy>
constexpr copy_n_t<ExPolicy> copy_n = copy_n_t<ExPolicy>{ }

template<typename ExPolicy>
struct copy_n_t : public hpx::functional::detail::tag_fallback<copy_n_t<ExPolicy>>
```

Friends

```
template<typename InIter, typename OutIter>
friend constexpr in_out_result<InIter, OutIter> tag_invoke (hpx::parallel::util::copy_n_t<
    InIter
    first,
    std::size_t
    count,
    OutIter
    dest)
```

namespace hpx

namespace parallel

namespace util

Variables

```
constexpr transform_loop_t transform_loop = transform_loop_t{ }

constexpr transform_loop_ind_t transform_loop_ind = transform_loop_ind_t{ }

template<typename ExPolicy>
constexpr transform_binary_loop_t<ExPolicy> transform_binary_loop = transform_binary_loop_t<ExPolicy>

template<typename ExPolicy>
constexpr transform_binary_loop_ind_t<ExPolicy> transform_binary_loop_ind = transform_binary_loop_ind_t<ExPolicy>

template<typename ExPolicy>
constexpr transform_loop_n_t<ExPolicy> transform_loop_n = transform_loop_n_t<ExPolicy>{ }
```

```
template<typename ExPolicy>
constexpr transform_loop_n_ind_t<ExPolicy> transform_loop_n_ind = transform_loop_n_ind_t<ExPolicy>

template<typename ExPolicy>
constexpr transform_binary_loop_n_t<ExPolicy> transform_binary_loop_n = transform_binary_loop_n_t<ExPolicy>

template<typename ExPolicy>
constexpr transform_binary_loop_ind_n_t<ExPolicy> transform_binary_loop_ind_n = transform_binary_loop_ind_n_t<ExPolicy>

template<typename ExPolicy>
struct transform_binary_loop_ind_n_t : public hpx::functional::detail::tag_fallback<transform_binary...
```

Friends

```
template<typename InIter1, typename InIter2, typename OutIter, typename F>
friend constexpr hpx::tuple<InIter1, InIter2, OutIter> tag_fallback_invoke (hpx::parallel::util::trans...
    InIter1,
    first1,
    std::size_t
    count,
    InIter2,
    first2,
    Out-
    Iter,
    dest,
    F
    &&f)
```

```
template<typename ExPolicy>
struct transform_binary_loop_ind_t : public hpx::functional::detail::tag_fallback<transform_binary...
```

Friends

```
template<typename InIter1B, typename InIter1E, typename InIter2, typename OutIter, typename F>
friend constexpr util::in_in_out_result<InIter1B, InIter2, OutIter> tag_fallback_invoke (hpx::parallel...
    InIter1B,
    first1,
    InIter1E,
    last1,
    InIter2,
    first2,
    Out-
    Iter,
    dest,
    F
    &&f)
```

```
template<typename InIter1B, typename InIter1E, typename InIter2B, typename InIter2E, typename ...
```

```
friend constexpr util::in_in_out_result<InIter1B, InIter2B, OutIter> tag_fallback_invoke (hpx::parallel::util::in_in_out_result<InIter1B, InIter2B, OutIter> first1, hpx::parallel::util::in_in_out_result<InIter1E, InIter2E, OutIter> last1, hpx::parallel::util::in_in_out_result<InIter2B, InIter2E, OutIter> first2, hpx::parallel::util::in_in_out_result<InIter2E, OutIter> last2, hpx::parallel::util::out_iterator<OutIter> dest, hpx::function<F> F, hpx::function<F> &&f)
```

```
template<typename ExPolicy>
struct transform_binary_loop_n_t : public hpx::functional::detail::tag_fallback<transform_binary_loop_n_t, ExPolicy>
```

Friends

```
template<typename InIter1, typename InIter2, typename OutIter, typename F>
friend constexpr hpx::tuple<InIter1, InIter2, OutIter> tag_fallback_invoke (hpx::parallel::util::transform<InIter1, InIter2, OutIter, F, std::size_t, count_t, InIter2> first1, hpx::parallel::util::transform<InIter2, OutIter, F, std::size_t, count_t, InIter1> first2, hpx::parallel::util::out_iterator<OutIter> dest, hpx::function<F> F, hpx::function<F> &&f)
```

```
template<typename ExPolicy>
struct transform_binary_loop_t : public hpx::functional::detail::tag_fallback<transform_binary_loop_t, ExPolicy>
```

Friends

```
template<typename InIter1B, typename InIter1E, typename InIter2, typename OutIter, typename F>
friend constexpr util::in_in_out_result<InIter1B, InIter2, OutIter> tag_fallback_invoke (hpx::parallel::util::in_in_out_result<InIter1B, InIter2, OutIter> first1, hpx::parallel::util::in_in_out_result<InIter1E, InIter2, OutIter> last1, hpx::parallel::util::in_in_out_result<InIter2, InIter2, OutIter> first2, hpx::parallel::util::in_in_out_result<InIter2, OutIter> last2, hpx::parallel::util::out_iterator<OutIter> dest, hpx::function<F> F, hpx::function<F> &&f)
```

```
template<typename InIter1B, typename InIter1E, typename InIter2B, typename InIter2E, typename OutIter>
friend constexpr util::in_in_out_result<InIter1B, InIter2B, OutIter> tag_fallback_invoke(hpx::parallel::util::transform_loop_n_t<InIter1B, InIter1E, InIter2B, InIter2E, OutIter> &f, InIter1B first1, InIter1E last1, InIter2B first2, InIter2E last2, OutIter dest, F &&f)

struct transform_loop_ind_t : public hpx::functional::detail::tag_fallback<transform_loop_ind_t>
```

Friends

```
template<typename ExPolicy, typename IterB, typename IterE, typename OutIter, typename F>
friend constexpr util::in_out_result<IterB, OutIter> tag_fallback_invoke(hpx::parallel::util::transform_loop_n_t<ExPolicy, IterB, IterE, OutIter, F> &f, IterB it, IterE end, OutIter dest, F &&f)

template<typename ExPolicy>
struct transform_loop_n_ind_t : public hpx::functional::detail::tag_fallback<transform_loop_n_ind_t<ExPolicy>>
```

Friends

```
template<typename Iter, typename OutIter, typename F>
friend constexpr std::pair<Iter, OutIter> tag_fallback_invoke(hpx::parallel::util::transform_loop_n_t<Iter, OutIter, F> &f, Iter it, std::size_t count, OutIter dest, F &&f)

template<typename ExPolicy>
struct transform_loop_n_t : public hpx::functional::detail::tag_fallback<transform_loop_n_t<ExPolicy>>
```

Friends

```
template<typename Iter, typename OutIter, typename F>
friend constexpr std::pair<Iter, OutIter> tag_fallback_invoke (hpx::parallel::util::transform_loop_n
    Iter           it,
    std::size_t     count,
    OutIter        dest, F &&f)

struct transform_loop_t : public hpx::functional::detail::tag_fallback<transform_loop_t>
```

Friends

```
template<typename ExPolicy, typename IterB, typename IterE, typename OutIter, typename F>
friend constexpr util::in_out_result<IterB, OutIter> tag_fallback_invoke (hpx::parallel::util::transfo
    Ex-
    Pol-
    icy&&,
    IterB
    it,
    IterE
    end,
    Out-
    Iter
    dest,
    F
    &&f)
```

allocator_support

See *Public API* for a list of names and headers that are part of the public HPX API.

Functions

```
void *__aligned_alloc (std::size_t alignment, std::size_t size)
void __aligned_free (void *p)
namespace hpx

namespace util
```

Functions

```
template<typename T>
constexpr bool operator==(aligned_allocator<T> const&, aligned_allocator<T> const&)

template<typename T>
constexpr bool operator!=(aligned_allocator<T> const&, aligned_allocator<T> const&)

template<typename T = int>
struct aligned_allocator
```

Public Types

```
typedef T value_type
typedef T *pointer
typedef T &reference
typedef T const &const_reference
typedef std::size_t size_type
typedef std::ptrdiff_t difference_type
typedef std::true_type is_always_equal
typedef std::true_type propagate_on_container_move_assignment
```

Public Functions

```
aligned_allocator()

template<typename U>
aligned_allocator(aligned_allocator<U> const&)

pointer address(reference x) const
const_pointer address(const_reference x) const
HPX_NODISCARD pointer hpx::util::aligned_allocator::allocate(size_type n, void const

void deallocate(pointer p, size_type)

size_type max_size() const

template<typename U, typename ...Args>
void construct(U *p, Args&&... args)

template<typename U>
void destroy(U *p)
```

Public Members

```
const typedef T* hpx::util::aligned_allocator::const_pointer  
template<typename U>  
struct rebind
```

Public Types

```
template<>  
typedef aligned_allocator<U> other  
namespace hpx  
  
namespace util  
  
template<typename Allocator>  
struct allocator_deleter
```

Public Functions

```
template<typename SharedState>  
void operator() (SharedState *state)
```

Public Members

```
Allocator alloc_  
namespace hpx  
  
namespace util
```

Typedefs

```
template<typename T = int>  
using internal_allocator = std::allocator<T>  
namespace hpx  
  
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_allocator_v = is_allocator<T>::value
```

asio

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

namespace util

TypeDefs

```
using endpoint_iterator_type = asio::ip::tcp::resolver::iterator
```

Functions

```
bool get_endpoint (std::string const &addr, std::uint16_t port, asio::ip::tcp::endpoint &ep)  
std::string get_endpoint_name (asio::ip::tcp::endpoint const &ep)  
asio::ip::tcp::endpoint resolve_hostname (std::string const &hostname, std::uint16_t port,  
asio::io_context &io_service)  
std::string resolve_public_ip_address ()  
std::string cleanup_ip_address (std::string const &addr)  
endpoint_iterator_type connect_begin (std::string const &address, std::uint16_t port,  
asio::io_context &io_service)  
template<typename Locality>  
endpoint_iterator_type connect_begin (Locality const &loc, asio::io_context &io_service)  
    Returns an iterator which when dereferenced will give an endpoint suitable for a call to connect()  
    related to this locality.  
endpoint_iterator_type connect_end ()  
endpoint_iterator_type accept_begin (std::string const &address, std::uint16_t port,  
asio::io_context &io_service)  
template<typename Locality>  
endpoint_iterator_type accept_begin (Locality const &loc, asio::io_context &io_service)  
    Returns an iterator which when dereferenced will give an endpoint suitable for a call to accept() related  
    to this locality.  
endpoint_iterator_type accept_end ()  
bool split_ip_address (std::string const &v, std::string &host, std::uint16_t &port)
```

namespace hpx

```
namespace util

    struct map_hostnames
```

Public Types

```
typedef util::function_nonser<std::string (std::string  
                                         transform_function_type  
                                         const&) >
```

Public Functions

```
map_hostnames (bool debug = false)
void use_suffix (std::string const &suffix)
void use_prefix (std::string const &prefix)
void use_transform (transform_function_type const &f)
std::string map (std::string host_name, std::uint16_t port) const
```

Private Members

```
transform_function_type transform_
std::string suffix_
std::string prefix_
bool debug_
```

assertion

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_ASSERT_CURRENT_FUNCTION
namespace hpx

    namespace assertion
```

Functions

```
std::ostream &operator<< (std::ostream &os, source_location const &loc)

struct source_location
#include <source_location.hpp> This contains the location information where HPX_ASSERT has been called
```

Public Members

```
const char *file_name
unsigned line_number
const char *function_name
```

Defines

HPX_ASSERT (*expr*)

This macro asserts that *expr* evaluates to true.

If *expr* evaluates to false, The source location and *msg* is being printed along with the expression and additional. Afterwards the program is being aborted. The assertion handler can be customized by calling `hpx::assertion::set_assertion_handler()`.

Parameters

- *expr*: The expression to assert on. This can either be an expression that's convertible to bool or a callable which returns bool
- *msg*: The optional message that is used to give further information if the assert fails. This should be convertible to a `std::string`

Asserts are enabled if *HPX_DEBUG* is set. This is the default for `CMAKE_BUILD_TYPE=Debug`

HPX_ASSERT_MSG (*expr, msg*)

See `HPX_ASSERT`

namespace hpx

namespace assertion

TypeDefs

```
using assertion_handler = void (*) (source_location const &loc, const char *expr,
                                         std::string const &msg)
```

The signature for an assertion handler.

Functions

```
void set_assertion_handler(assertion_handler handler)
```

Set the assertion handler to be used within a program. If the handler has been set already once, the call to this function will be ignored.

Note This function is not thread safe

async_base

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

Functions

```
template<typename F, typename ...Ts>
bool apply(F &&f, Ts&&... ts)
```

```
namespace hpx
```

Functions

```
template<typename F, typename ...Ts>
decltype(auto) async(F &&f, Ts&&... ts)
```

```
namespace hpx
```

Functions

```
template<typename F, typename ...Ts>
auto dataflow(F &&f, Ts&&... ts)
```

```
HPX_FORWARD(F, f)
```

```
HPX_FORWARD(Ts, ts)
```

```
template<typename Allocator, typename F, typename ...Ts>
auto dataflow_alloc(Allocator const &alloc, F &&f, Ts&&... ts)
```

```
namespace hpx
```

```
struct launch : public detail::policy_holder<>
```

#include <launch_policy.hpp> Launch policies for *hpx::async* etc.

Public Functions

constexpr launch()

Default constructor. This creates a launch policy representing all possible launch modes

constexpr launch(detail::async_policy p)

Create a launch policy representing asynchronous execution.

constexpr launch(detail::fork_policy p)

Create a launch policy representing asynchronous execution. The new thread is executed in a preferred way

constexpr launch(detail::sync_policy p)

Create a launch policy representing synchronous execution.

constexpr launch(detail::deferred_policy p)

Create a launch policy representing deferred execution.

constexpr launch(detail::apply_policy p)

Create a launch policy representing fire and forget execution.

template<typename F>

constexpr launch(detail::select_policy<F> const &p)

Create a launch policy representing fire and forget execution.

template<typename Launch, typename Enable = std::enable_if_t<hpx::traits::is_launch_policy_v<Launch>>>

constexpr launch(Launch l, threads::thread_priority priority, threads::thread_stacksize stack-size, threads::thread_schedule_hint hint)

Public Static Attributes

const detail::async_policy async

Predefined launch policy representing asynchronous execution.

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.

Friends

```

launch tag_invoke(hpx::execution::experimental::with_priority_t, launch const &policy,
                    threads::thread_priority priority)
friend constexpr hpx::threads::thread_priority tag_invoke(hpx::execution::experimental::get_priority_t,
                                                               launch const &policy)
launch tag_invoke(hpx::execution::experimental::with_stacksize_t, launch const &policy,
                     threads::thread_stacksize stacksize)
friend constexpr hpx::threads::thread_stacksize tag_invoke(hpx::execution::experimental::get_stacksize_t,
                                                               launch const &policy)
launch tag_invoke(hpx::execution::experimental::with_hint_t, launch const &policy,
                     threads::thread_schedule_hint hint)
friend constexpr hpx::threads::thread_schedule_hint tag_invoke(hpx::execution::experimental::get_hint_t,
                                                               launch const &policy)
namespace hpx

namespace execution

namespace experimental

```

Variables

```

hpx::execution::experimental::with_priority_t with_priority
hpx::execution::experimental::get_priority_t get_priority
hpx::execution::experimental::with_stacksize_t with_stacksize
hpx::execution::experimental::get_stacksize_t get_stacksize
hpx::execution::experimental::with_hint_t with_hint
hpx::execution::experimental::get_hint_t get_hint
hpx::execution::experimental::with_annotation_t with_annotation
hpx::execution::experimental::get_annotation_t get_annotation
namespace hpx

```

Functions

```

template<typename F, typename ...Ts>
auto sync(F &&f, Ts&&... ts)
namespace hpx

namespace traits

```

Variables

```
template<typename Policy>
constexpr bool is_launch_policy_v = is_launch_policy<Policy>::value
```

async_combinators

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace lcos
```

Functions

```
template<typename Future, typename F>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::")
```

The one argument version is special in the sense that it returns the expected value directly (without wrapping it into a tuple).

```
namespace hpx
```

Functions

```
template<typename ...Ts>
```

```
tuple<future<Ts>...> split_future(future<tuple<Ts...>> &&f)
```

The function *split_future* is an operator allowing to split a given future of a sequence of values (any tuple, std::pair, or std::array) into an equivalent container of futures where each future represents one of the values from the original future. In some sense this function provides the inverse operation of *when_all*.

Return Returns an equivalent container (same container type as passed as the argument) of futures, where each future refers to the corresponding value in the input parameter. All of the returned futures become ready once the input future has become ready. If the input future is exceptional, all output futures will be exceptional as well.

Note The following cases are special:

```
tuple<future<void> > split_future(future<tuple<>> && f);
array<future<void>, 1> split_future(future<array<T, 0>> && f);
```

here the returned futures are directly representing the futures which were passed to the function.

Parameters

- *f*: [in] A future holding an arbitrary sequence of values stored in a tuple-like container. This facility supports *hpx::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

```
namespace hpx
```

Functions

```
template<typename InputIter>
void wait_all(InputIter first, InputIter last)
```

The function `wait_all` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note The function `wait_all` returns after all futures have become ready. All input futures are still valid after `wait_all` returns.

Note The function `wait_all` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_all_nothrow` instead.

Parameters

- `first`: The iterator pointing to the first element of a sequence of `future` or `shared_future` objects for which `wait_all` should wait.
- `last`: The iterator pointing to the last element of a sequence of `future` or `shared_future` objects for which `wait_all` should wait.

```
template<typename R>
void wait_all(std::vector<future<R>> &&futures)
```

The function `wait_all` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note The function `wait_all` returns after all futures have become ready. All input futures are still valid after `wait_all` returns.

Note The function `wait_all` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_all_nothrow` instead.

Parameters

- `futures`: A vector or array holding an arbitrary amount of `future` or `shared_future` objects for which `wait_all` should wait.

```
template<typename R, std::size_t N>
void wait_all(std::array<future<R>, N> &&futures)
```

The function `wait_all` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note The function `wait_all` returns after all futures have become ready. All input futures are still valid after `wait_all` returns.

Note The function `wait_all` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_all_nothrow` instead.

Parameters

- `futures`: A vector or array holding an arbitrary amount of `future` or `shared_future` objects for which `wait_all` should wait.

```
template<typename ...T>
void wait_all(T&&... futures)
```

The function `wait_all` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note The function `wait_all` returns after all futures have become ready. All input futures are still valid after `wait_all` returns.

Note The function `wait_all` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_all_nothrow` instead.

Parameters

- `futures`: An arbitrary number of `future` or `shared_future` objects, possibly holding different types for which `wait_all` should wait.

```
template<typename InputIter>
void wait_all_n(InputIter begin, std::size_t count)
```

The function `wait_all_n` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

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.

Note The function `wait_all_n` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_all_n_nothrow` instead.

Parameters

- `begin`: The iterator pointing to the first element of a sequence of `future` or `shared_future` objects for which `wait_all_n` should wait.
- `count`: The number of elements in the sequence starting at `first`.

```
namespace hpx
```

Functions

```
template<typename InputIter>
void wait_any(InputIter first, InputIter last)
```

The function `wait_any` is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function `wait_any` returns after at least one future has become ready. All input futures are still valid after `wait_any` returns.

Note The function `wait_any` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_any_nothrow` instead.

Parameters

- `first`: [in] The iterator pointing to the first element of a sequence of `future` or `shared_future` objects for which `wait_any` should wait.
- `last`: [in] The iterator pointing to the last element of a sequence of `future` or `shared_future` objects for which `wait_any` should wait.

```
template<typename R>
void wait_any (std::vector<future<R>> &futures)
```

The function `wait_any` is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function `wait_any` returns after at least one future has become ready. All input futures are still valid after `wait_any` returns.

Note The function `wait_any` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_any_nothrow` instead.

Parameters

- `futures`: [in] A vector holding an arbitrary amount of `future` or `shared_future` objects for which `wait_any` should wait.

```
template<typename R, std::size_t N>
void wait_any (std::array<future<R>, N> &futures)
```

The function `wait_any` is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function `wait_any` returns after at least one future has become ready. All input futures are still valid after `wait_any` returns.

Note The function `wait_any` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_any_nothrow` instead.

Parameters

- `futures`: [in] An array holding an arbitrary amount of `future` or `shared_future` objects for which `wait_any` should wait.

```
template<typename ...T>
void wait_any (T&&... futures)
```

The function `wait_any` is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function `wait_any` returns after at least one future has become ready. All input futures are still valid after `wait_any` returns.

Note The function `wait_any` will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use `wait_any_nothrow` instead.

Parameters

- `futures`: [in] An arbitrary number of `future` or `shared_future` objects, possibly holding different types for which `wait_any` should wait.

```
template<typename InputIter>
void wait_any_n(InputIter first, std::size_t count)
```

The function *wait_any_n* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function *wait_any_n* returns after at least one future has become ready. All input futures are still valid after *wait_any_n* returns.

Note The function *wait_any_n* will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_any_n_nothrow* instead.

Parameters

- *first*: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_any_n* should wait.
- *count*: [in] The number of elements in the sequence starting at *first*.

```
namespace hpx
```

Functions

```
template<typename F, typename Future>
void wait_each(F &&f, std::vector<Future> &&futures)
```

The function *wait_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready. *wait_each* returns after all futures have been become ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- *f*: The function which will be called for each of the input futures once the future has become ready.
- *futures*: A vector holding an arbitrary amount of *future* or *shared_future* objects for which *wait_each* should wait.

```
template<typename F, typename Iterator>
void wait_each(F &&f, Iterator begin, Iterator end)
```

The function *wait_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready. *wait_each* returns after all futures have been become ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- *f*: The function which will be called for each of the input futures once the future has become ready.
- *begin*: The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.
- *end*: The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.

```
template<typename F, typename ...T>
void wait_each(F &&f, T&&... futures)
```

The function *wait_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready. *wait_each* returns after all futures have been become ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- *f*: The function which will be called for each of the input futures once the future has become ready.
- *futures*: An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_each* should wait.

```
template<typename F, typename Iterator>
void wait_each_n(F &&f, Iterator begin, std::size_t count)
```

The function *wait_each* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- *f*: The function which will be called for each of the input futures once the future has become ready.
- *begin*: The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_each_n* should wait.
- *count*: The number of elements in the sequence starting at *first*.

```
namespace hpx
```

Functions

```
template<typename InputIter>
void wait_some(std::size_t n, InputIter first, InputIter last)
```

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after *n* of them finished executing.

Note The function *wait_some* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note The function *wait_some* will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- *n*: [in] The number of futures out of the arguments which have to become ready in order for the function to return.
- *first*: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- *last*: [in] The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.

```
template<typename R>
void wait_some(std::size_t n, std::vector<future<R>> &&futures)
```

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after *n* of them finished executing.

Note The function *wait_some* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note The function *wait_some* will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- *n*: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- *futures*: [in] A vector holding an arbitrary amount of *future* or *shared_future* objects for which *wait_some* should wait.

```
template<typename R, std::size_t N>
void wait_some(std::size_t n, std::array<future<R>, N> &&futures)
```

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after *n* of them finished executing.

Note The function *wait_some* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note The function *wait_some* will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `futures`: [in] An array holding an arbitrary amount of *future* or *shared_future* objects for which *wait_some* should wait.

```
template<typename ...T>
void wait_some (std::size_t n, T&&... futures)
```

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after *n* of them finished executing.

Note The function *wait_all* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note The function *wait_some* will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `futures`: [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_some* should wait.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
template<typename InputIter>
void wait_some_n (std::size_t n, InputIter first, std::size_t count)
```

The function *wait_some_n* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after *n* of them finished executing.

Note The function *wait_some_n* returns after *n* futures have become ready. All input futures are still valid after *wait_some_n* returns.

Note The function *wait_some_n* will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_n_nothrow* instead.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `first`: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- `count`: [in] The number of elements in the sequence starting at *first*.

```
namespace hpx
```

Functions

```
template<typename InputIter, typename Container = vector<future<typename std::iterator_traits<InputIter>::value_type>>
hpx::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>
hpx::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>
hpx::future<hpx::tuple<hpx::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>>
hpx::future<Container> when_all_n (InputIter begin, std::size_t count)
```

The function `when_all_n` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after they finished executing.

Return Returns a future holding the same list of futures as has been passed to `when_all_n`.

- `future<Container<future<R>>>`: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output vector will be the same as given by the input iterator.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note None of the futures in the input sequence are invalidated.

Parameters

- `begin`: [in] The iterator pointing to the first element of a sequence of `future` or `shared_future` objects for which `wait_all_n` should wait.
- `count`: [in] The number of elements in the sequence starting at `first`.

Exceptions

- `This`: function will throw errors which are encountered while setting up the requested operation only. Errors encountered while executing the operations delivering the results to be stored in the futures are reported through the futures themselves.

```
namespace hpx
```

Functions

```
template<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 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*

`namespace hpx`

Functions

```
template<typename F, typename Future>
future<void> when_each (F &&f, std::vector<Future> &&futures)
The function when_each is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.
```

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that `std::size_t` is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Return Returns a future representing the event of all input futures being ready.

Parameters

- `f`: The function which will be called for each of the input futures once the future has become ready.
- `futures`: A vector holding an arbitrary amount of *future* or *shared_future* objects for which *wait_each* should wait.

`template<typename F, typename Iterator>`

```
future<Iterator> when_each (F &&f, Iterator begin, Iterator end)
```

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Return Returns a future representing the event of all input futures being ready.

Parameters

- *f*: The function which will be called for each of the input futures once the future has become ready.
- *begin*: The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.
- *end*: The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.

```
template<typename F, typename ...Ts>
future<void> when_each (F &&f, Ts&&... futures)
```

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Return Returns a future representing the event of all input futures being ready.

Parameters

- *f*: The function which will be called for each of the input futures once the future has become ready.
- *futures*: An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_each* should wait.

```
template<typename F, typename Iterator>
future<Iterator> when_each_n (F &&f, Iterator begin, std::size_t count)
```

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Return Returns a future holding the iterator pointing to the first element after the last one.

Parameters

- `f`: The function which will be called for each of the input futures once the future has become ready.
- `begin`: The iterator pointing to the first element of a sequence of `future` or `shared_future` objects for which `wait_each_n` should wait.
- `count`: The number of elements in the sequence starting at `first`.

```
namespace hpx
```

Functions

template<typename **InputIter**, typename **Container** = vector<future<typename std::iterator_traits<**InputIter**>::value_type>>
future<when_some_result<Container>> **when_some** (std::size_t *n*, Iterator *first*, Iterator *last*)

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.

template<typename **Range**>

future<when_some_result<Range>> **when_some** (std::size_t *n*, Range &&futures)

The function `when_some` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after *n* of them finished executing.

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.

template<typename ...`Ts`>

`future<when_some_result<tuple<future<T>...>>> when_some (std::size_t n, Ts&&... futures)`

The function `when_some` is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after `n` of them finished executing.

Note The future returned by the function `when_some` becomes ready when at least `n` argument futures have become ready.

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)`

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`.

```
template<typename Sequence>
struct when_some_result
#include <when_some.hpp> Result type for when_some, contains a sequence of futures and indices pointing to ready futures.
```

Public Members

`std::vector<std::size_t> indices`
List of indices of futures that have become ready.

Sequence `futures`
The sequence of futures as passed to `hpx::when_some`.

async_cuda

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

`namespace cuda`

`namespace experimental`

`struct cuda_event_pool`

Public Functions

```
cuda_event_pool()  
~cuda_event_pool()  
bool pop(cudaEvent_t &event)  
bool push(cudaEvent_t event)
```

Public Static Functions

```
static cuda_event_pool &get_event_pool()
```

Public Static Attributes

```
constexpr int initial_events_in_pool = 128
```

Private Functions

```
void add_event_to_pool()
```

Private Members

```
boost::lockfree::stack<cudaEvent_t, boost::lockfree::fixed_sized<false>> free_list_  
namespace hpx  
  
namespace cuda  
  
namespace experimental
```

Functions

```
void check_cuda_error(cudaError_t err)  
struct cuda_exception : public exception
```

Public Functions

```
cuda_exception(const std::string &msg, cudaError_t err)  
cudaError_t get_cuda_errorcode()
```

Protected Attributes

```
cudaError_t err_

namespace hpx

namespace cuda

namespace experimental

struct cuda_executor : public hpx::cuda::experimental::cuda_executor_base
```

Public Functions

```
cuda_executor(std::size_t device, bool event_mode = true)

~cuda_executor()

template<typename F, typename ...Ts>
decltype(auto) post(F &&f, Ts&&... ts)

template<typename F, typename ...Ts>
decltype(auto) async_execute(F &&f, Ts&&... ts)
```

Protected Functions

```
template<typename R, typename ...Params, typename ...Args>
void apply(R (*cuda_function)) Params...
    , Args&&... args

template<typename R, typename ...Params, typename ...Args>
hpx::future<void> async(R (*cuda_kernel)) Params...
    , Args&&... args

struct cuda_executor_base
Subclassed by hpx::cuda::experimental::cuda_executor
```

Public Types

```
using future_type = hpx::future<void>
```

Public Functions

```
cuda_executor_base(std::size_t device, bool event_mode)

future_type get_future()
```

Protected Attributes

```
int device_
bool event_mode_
cudaStream_t stream_
std::shared_ptr<hpx::cuda::experimental::target> target_

namespace hpx

namespace cuda

namespace experimental
```

Typedefs

```
using event_mode = std::true_type
using callback_mode = std::false_type

namespace hpx

namespace cuda

namespace experimental

struct enable_user_polling
```

Public Functions

```
enable_user_polling(std::string const &pool_name = "")
~enable_user_polling()
```

Private Members

```
std::string pool_name_

namespace hpx

namespace cuda

namespace experimental
```

Functions

```
std::vector<target> get_local_targets()
void print_local_targets()

namespace hpx
```

```
namespace compute
```

```
namespace cuda
```

TypeDefs

```
using instead = hpx::cuda::experimental::target
namespace cuda
```

```
namespace experimental
```

Functions

```
target &get_default_target()
```

```
struct target
```

Public Functions

```
target()
```

```
target(int device)
```

```
target(target const &rhs)
```

```
target(target &&rhs)
```

```
target &operator=(target const &rhs)
```

```
target &operator=(target &&rhs)
```

```
native_handle_type &native_handle()
```

```
native_handle_type const &native_handle() const
```

```
void synchronize() const
```

```
hpx::future<void> get_future_with_event() const
```

```
hpx::future<void> get_future_with_callback() const
```

```
template<typename Allocator>
```

```
hpx::future<void> get_future_with_event(Allocator const &alloc) const
```

```
template<typename Allocator>
```

```
hpx::future<void> get_future_with_callback(Allocator const &alloc) const
```

Public Static Functions

```
static std::vector<target> get_local_targets()
```

Private Members

```
native_handle_type handle_
```

Friends

```
bool operator==(target const &lhs, target const &rhs)
```

```
struct native_handle_type
```

Public Types

```
typedef hpx::lcos::local::spinlock mutex_type
```

Public Functions

```
native_handle_type(int device = 0)
```

```
~native_handle_type()
```

```
native_handle_type(native_handle_type const &rhs)
```

```
native_handle_type(native_handle_type &&rhs)
```

```
native_handle_type &operator=(native_handle_type const &rhs)
```

```
native_handle_type &operator=(native_handle_type &&rhs)
```

```
cudaStream_t get_stream() const
```

```
int get_device() const
```

```
std::size_t processing_units() const
```

```
std::size_t processor_family() const
```

```
std::string processor_name() const
```

```
void reset()
```

Private Functions

```
void init_processing_units()
```

Private Members

```
mutex_type mtx_
int device_
std::size_t processing_units_
std::size_t processor_family_
std::string processor_name_
cudaStream_t stream_
```

Friends

```
friend hpx::cuda::experimental::target
namespace hpx
```

```
namespace cuda
```

```
namespace experimental
```

Variables

```
hpx::cuda::experimental::transform_stream_t transform_stream
struct transform_stream_t : public hpx::functional::detail::tag_fallback<transform_stream_t>
```

Friends

```
template<typename S, typename F, typename = std::enable_if_t<!std::is_same<std::decay_t<F>, cudaStream_t>>
friend constexpr auto tagFallbackInvoke(transform_stream_t, S &&s, F
                                         &&f, cudaStream_t stream = {})

template<typename F>
friend constexpr auto tagFallbackInvoke(transform_stream_t, F &&f, cudaS-
                                         tream_t stream = {})
```

async_local

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

Functions

```
template<typename Action, typename F, typename ...Ts>
auto async (F &&f, Ts&&... ts)
```

```
namespace hpx
```

Functions

```
template<typename Action, typename F, typename ...Ts>
auto sync (F &&f, Ts&&... ts)
```

```
async_mpi
```

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace mpi
```

```
namespace experimental
```

```
struct mpi_exception : public exception
```

Public Functions

```
mpi_exception (int err_code, const std::string &msg = "")
int get_mpi_errorcode ()
```

Protected Attributes

```
int err_code_
```

```
namespace hpx
```

```
namespace mpi
```

```
namespace experimental
```

```
struct executor
```

Public Types

```
using execution_category = hpx::execution::parallel_execution_tag
using executor_parameters_type = hpx::execution::static_chunk_size
```

Public Functions

```
constexpr executor (MPI_Comm communicator = MPI_COMM_WORLD)

template<typename F, typename ...Ts>
decltype(auto) async_execute (F &&f, Ts&&... ts) const

std::size_t in_flight_estimate () const
```

Private Members

```
MPI_Comm communicator_

namespace hpx

namespace mpi

namespace experimental
```

TypeDefs

```
using print_on = debug::enable_print<false>
```

Functions

```
static constexpr print_on hpx::mpi::experimental::mpi_debug ("MPI_FUT")

void set_error_handler ()

hpx::future<void> get_future (MPI_Request request)

hpx::threads::policies::detail::polling_status poll ()

void wait ()

template<typename F>
void wait (F &&f)

void init (bool init_mpi = false, std::string const &pool_name = "", bool init_errorhandler =
    false)

void finalize (std::string const &pool_name = "")

template<typename ...Args>
void debug (Args&&... args)

struct enable_user_polling
```

Public Functions

```
enable_user_polling(std::string const &pool_name = "", bool init_errorhandler =
    false)
~enable_user_polling()
```

Private Members

```
std::string pool_name_
namespace hpx
namespace mpi
namespace experimental
```

Variables

```
hpx::mpi::experimental::transform_mpi_t transform_mpi
struct transform_mpi_t : public hpx::functional::detail::tagFallback<transform_mpi_t>
```

Friends

```
template<typename Sender, typename F>
friend constexpr auto tagFallback_invoke(transform_mpi_t, Sender &&s, F
    &&f)
template<typename F>
friend constexpr auto tagFallback_invoke(transform_mpi_t, F &&f)
```

batch_environments

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

```
namespace batch_environments
struct alps_environment
```

Public Functions

```
alps_environment (std::vector<std::string> &nodelist, bool debug)
bool valid() const
std::size_t node_num() const
std::size_t num_threads() const
std::size_t num_localities() const
```

Private Members

```
std::size_t node_num_
std::size_t num_threads_
std::size_t num_localities_
bool valid_
```

```
namespace hpx
```

```
namespace util
```

```
struct batch_environment
```

Public Types

```
typedef std::map<asio::ip::tcp::endpoint, std::pair<std::string, std::size_t>> node_map_type
```

Public Functions

```
batch_environment (std::vector<std::string> &nodelist, bool have_mpi = false, bool debug = false, bool enable = true)
std::string init_from_nodelist (std::vector<std::string> const &nodes, std::string const &agas_host)
std::size_t retrieve_number_of_threads() const
std::size_t retrieve_number_of_localities() const
std::size_t retrieve_node_number() const
std::string host_name() const
std::string host_name (std::string const &def_hpx_name) const
std::string agas_host_name (std::string const &def_agas) const
std::size_t agas_node() const
bool found_batch_environment() const
std::string get_batch_name() const
```

Public Members

```
std::string agas_node_
std::size_t agas_node_num_
std::size_t node_num_
std::size_t num_threads_
node_map_type nodes_
std::size_t num_localities_
std::string batch_name_
bool debug_
```

```
namespace hpx
```

```
namespace util
```

```
namespace batch_environments
```

```
struct pbs_environment
```

Public Functions

```
pbs_environment (std::vector<std::string> &nodelist, bool have_mpi, bool debug)
bool valid() const
std::size_t node_num() const
std::size_t num_threads() const
std::size_t num_localities() const
```

Private Functions

```
void read_nodefile (std::vector<std::string> &nodelist, bool have_mpi, bool debug)
void read_nodelist (std::vector<std::string> &nodelist, bool debug)
```

Private Members

```
std::size_t node_num_
std::size_t num_localities_
std::size_t num_threads_
bool valid_
```

```
namespace hpx
```

```
namespace util

namespace batch_environments

struct slurm_environment
```

Public Functions

```
slurm_environment (std::vector<std::string> &nodelist, bool debug)
bool valid() const
std::size_t node_num() const
std::size_t num_threads() const
std::size_t num_localities() const
```

Private Functions

```
void retrieve_number_of_localities (bool debug)
void retrieve_number_of_tasks (bool debug)
void retrieve_nodelist (std::vector<std::string> &nodes, bool debug)
void retrieve_number_of_threads ()
```

Private Members

```
std::size_t node_num_
std::size_t num_threads_
std::size_t num_tasks_
std::size_t num_localities_
bool valid_
```

cache

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

```
namespace util
```

```
namespace cache
```

```
template<typename Key, typename Entry, typename UpdatePolicy = std::less<Entry>, typename InsertPolicy = std::less<Entry>, typename Compare = std::less<Key>>
```

```
class local_cache
#include <hpx/cache/local_cache.hpp> The local_cache implements the basic functionality
needed for a local (non-distributed) cache.
```

Template Parameters

- Key: The type of the keys to use to identify the entries stored in the cache
- Entry: The type of the items to be held in the cache, must model the CacheEntry concept
- UpdatePolicy: A (optional) type specifying a (binary) function object used to sort the cache entries based on their ‘age’. The ‘oldest’ entries (according to this sorting criteria) will be discarded first if the maximum capacity of the cache is reached. The default is std::less<Entry>. The function object will be invoked using 2 entry instances of the type Entry. This type must model the UpdatePolicy model.
- InsertPolicy: A (optional) type specifying a (unary) function object used to allow global decisions whether a particular entry should be added to the cache or not. The default is policies::always, imposing no global insert related criteria on the cache. The function object will be invoked using the entry instance to be inserted into the cache. This type must model the InsertPolicy model.
- CacheStorage: A (optional) container type used to store the cache items. The container must be an associative and STL compatible container. The default is a std::map<Key, Entry>.
- Statistics: A (optional) type allowing to collect some basic statistics about the operation of the cache instance. The type must conform to the CacheStatistics concept. The default value is the type statistics::no_statistics which does not collect any numbers, but provides empty stubs allowing the code to compile.

Public Types

```
typedef Key key_type
typedef Entry entry_type
typedef UpdatePolicy update_policy_type
typedef InsertPolicy insert_policy_type
typedef CacheStorage storage_type
typedef Statistics statistics_type
typedef entry_type::value_type value_type
typedef storage_type::size_type size_type
typedef storage_type::value_type storage_value_type
```

Public Functions

```
local_cache(size_type max_size = 0, update_policy_type const &up = update_policy_type(),
            insert_policy_type const &ip = insert_policy_type())
Construct an instance of a local_cache.
```

Parameters

- max_size: [in] The maximal size this cache is allowed to reach any time. The default is zero (no size limitation). The unit of this value is usually determined by the unit of the values returned by the entry’s get_size function.
- up: [in] An instance of the UpdatePolicy to use for this cache. The default is to use a default constructed instance of the type as defined by the UpdatePolicy template parameter.

- `ip`: [in] An instance of the `InsertPolicy` to use for this cache. The default is to use a default constructed instance of the type as defined by the `InsertPolicy` template parameter.

local_cache (`local_cache &&other`)

`size_type size() const`

Return current size of the cache.

Return The current size of this cache instance.

`size_type capacity() const`

Access the maximum size the cache is allowed to grow to.

Note The unit of this value is usually determined by the unit of the return values of the entry's function `entry::get_size`.

Return The maximum size this cache instance is currently allowed to reach. If this number is zero the cache has no limitation with regard to a maximum size.

`bool reserve(size_type max_size)`

Change the maximum size this cache can grow to.

Return This function returns `true` if successful. It returns `false` if the new `max_size` is smaller than the current limit and the cache could not be shrunk to the new maximum size.

Parameters

- `max_size`: [in] The new maximum size this cache will be allowed to grow to.

bool holds_key(key_type const &k) const

Check whether the cache currently holds an entry identified by the given key.

Note This function does not call the entry's function `entry::touch`. It just checks if the cache contains an entry corresponding to the given key.

Return This function returns `true` if the cache holds the referenced entry, otherwise it returns `false`.

Parameters

- `k`: [in] The key for the entry which should be looked up in the cache.

bool get_entry(key_type const &k, key_type &realkey, entry_type &val)

Get a specific entry identified by the given key.

Note The function will call the entry's `entry::touch` function if the value corresponding to the provided key is found in the cache.

Return This function returns `true` if the cache holds the referenced entry, otherwise it returns `false`.

Parameters

- `k`: [in] The key for the entry which should be retrieved from the cache.
- `val`: [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

bool get_entry(key_type const &k, entry_type &val)

Get a specific entry identified by the given key.

Note The function will call the entry's `entry::touch` function if the value corresponding to the provided key is found in the cache.

Return This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

Parameters

- *k*: [in] The key for the entry which should be retrieved from the cache.
- *val*: [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

bool **get_entry** (*key_type const &k*, *value_type &val*)

Get a specific entry identified by the given key.

Note The function will call the entry's *entry::touch* function if the value corresponding to the provided is found in the cache.

Return This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

Parameters

- *k*: [in] The key for the entry which should be retrieved from the cache
- *val*: [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding value.

bool **insert** (*key_type const &k*, *value_type const &val*)

Insert a new element into this cache.

Note This function invokes both, the insert policy as provided to the constructor and the function *entry::insert* of the newly constructed entry instance. If either of these functions returns false the key/value pair doesn't get inserted into the cache and the *insert* function will return *false*. Other reasons for this function to fail (return *false*) are a) the key/value pair is already held in the cache or b) inserting the new value into the cache maxed out its capacity and it was not possible to free any of the existing entries.

Return This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

Parameters

- *k*: [in] The key for the entry which should be added to the cache.
- *value*: [in] The value which should be added to the cache.

bool **insert** (*key_type const &k*, *entry_type &e*)

Insert a new entry into this cache.

Note This function invokes both, the insert policy as provided to the constructor and the function *entry::insert* of the provided entry instance. If either of these functions returns false the key/value pair doesn't get inserted into the cache and the *insert* function will return *false*. Other reasons for this function to fail (return *false*) are a) the key/value pair is already held in the cache or b) inserting the new value into the cache maxed out its capacity and it was not possible to free any of the existing entries.

Return This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

Parameters

- *k*: [in] The key for the entry which should be added to the cache.
- *value*: [in] The entry which should be added to the cache.

bool **update** (*key_type const &k*, *value_type const &val*)

Update an existing element in this cache.

Note The function will call the entry's *entry::touch* function if the indexed value is found in the cache.

Note The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Return This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

Parameters

- *k*: [in] The key for the value which should be updated in the cache.
- *value*: [in] The value which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.

```
template<typename F>
```

```
bool update_if (key_type const &k, value_type const &val, F f)
```

Update an existing element in this cache.

Note The function will call the entry's *entry::touch* function if the indexed value is found in the cache.

Note The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Return This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

Parameters

- *k*: [in] The key for the value which should be updated in the cache.
- *value*: [in] The value which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.
- *f*: [in] A callable taking two arguments, *k* and the key found in the cache (in that order). If *f* returns true, then the update will continue. If *f* returns false, then the update will not succeed.

```
bool update (key_type const &k, entry_type &e)
```

Update an existing entry in this cache.

Note The function will call the entry's *entry::touch* function if the indexed value is found in the cache.

Note The difference to the other overload of the *insert* function is that this overload replaces the whole cache entry, while the other overload replaces the cached value only, leaving the cache entry properties untouched.

Return This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

Parameters

- *k*: [in] The key for the entry which should be updated in the cache.
- *value*: [in] The entry which should be used as a replacement for the existing entry in the cache. Any existing entry is first removed and then this entry is added.

```
template<typename Func>
```

```
size_type erase (Func const &ep = policies::always<storage_value_type>())
```

Remove stored entries from the cache for which the supplied function object returns true.

Return This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry*::*get_size* functions of the removed entries).

Parameters

- *ep*: [in] This parameter has to be a (unary) function object. It is invoked for each of the entries currently held in the cache. An entry is considered for removal from the cache whenever the value returned from this invocation is *true*. Even then the entry might not be removed from the cache as its *entry*::*remove* function might return false.

size_type **erase** ()

Remove all stored entries from the cache.

Note All entries are considered for removal, but in the end an entry might not be removed from the cache as its *entry*::*remove* function might return false. This function is very useful for instance in conjunction with an entry's *entry*::*remove* function enforcing additional criteria like entry expiration, etc.

Return This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry*::*get_size* functions of the removed entries).

void **clear** ()

Clear the cache.

Unconditionally removes all stored entries from the cache.

statistics_type **const &get_statistics () const**

Allow to access the embedded statistics instance.

Return This function returns a reference to the statistics instance embedded inside this cache

statistics_type &get_statistics ()

Protected Functions

bool **free_space** (long *num_free*)

Private Types

typedef *storage_type*::iterator **iterator**

typedef *storage_type*::const_iterator **const_iterator**

typedef *std*::deque<*iterator*> **heap_type**

typedef *heap_type*::iterator **heap_iterator**

typedef *adapt*<UpdatePolicy, iterator> **adapted_update_policy_type**

typedef *statistics_type*::update_on_exit **update_on_exit**

Private Members

```
size_type max_size_
size_type current_size_
storage_type store_
heap_type entry_heap_
adapted_update_policy_type update_policy_
insert_policy_type insert_policy_
statistics_type statistics_

template<typename Func, typename Iterator>
struct adapt
```

Public Functions

```
template<>
adapt (Func f)
template<>
bool operator() (Iterator const &lhs, Iterator const &rhs) const
```

Public Members

```
template<>
Func f_
```

namespace hpx

namespace util

namespace cache

```
template<typename Key, typename Entry, typename Statistics = statistics::no_statisticsclass lru_cache
#include <hpx/cache/lru_cache.hpp> The lru_cache implements the basic functionality needed
for a local (non-distributed) LRU cache.
```

Template Parameters

- **Key**: The type of the keys to use to identify the entries stored in the cache
- **Entry**: The type of the items to be held in the cache.
- **Statistics**: A (optional) type allowing to collect some basic statistics about the operation of the cache instance. The type must conform to the CacheStatistics concept. The default value is the type *statistics::no_statistics* which does not collect any numbers, but provides empty stubs allowing the code to compile.

Public Types

```
typedef Key key_type
typedef Entry entry_type
typedef Statistics statistics_type
typedef std::pair<key_type, entry_type> entry_pair
typedef std::list<entry_pair> storage_type
typedef std::map<Key, typename storage_type::iterator> map_type
typedef std::size_t size_type
```

Public Functions

lru_cache (*size_type max_size = 0*)
Construct an instance of a *lru_cache*.

Parameters

- *max_size*: [in] The maximal size this cache is allowed to reach any time. The default is zero (no size limitation). The unit of this value is usually determined by the unit of the values returned by the entry's *get_size* function.

lru_cache (*lru_cache &&other*)

size_type size () const
Return current size of the cache.

Return The current size of this cache instance.

size_type capacity () const
Access the maximum size the cache is allowed to grow to.

Note The unit of this value is usually determined by the unit of the return values of the entry's function *entry::get_size*.

Return The maximum size this cache instance is currently allowed to reach. If this number is zero the cache has no limitation with regard to a maximum size.

void reserve (*size_type max_size*)
Change the maximum size this cache can grow to.

Parameters

- *max_size*: [in] The new maximum size this cache will be allowed to grow to.

bool holds_key (*key_type const &key*)
Check whether the cache currently holds an entry identified by the given key.

Note This function does not call the entry's function *entry::touch*. It just checks if the cache contains an entry corresponding to the given key.

Return This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

Parameters

- `k`: [in] The key for the entry which should be looked up in the cache.

```
bool get_entry (key_type const &key, key_type &realkey, entry_type &entry)  
Get a specific entry identified by the given key.
```

Note The function will “touch” the entry and mark it as recently used if the key was found in the cache.

Return This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

Parameters

- `key`: [in] The key for the entry which should be retrieved from the cache.
- `entry`: [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

```
bool get_entry (key_type const &key, entry_type &entry)  
Get a specific entry identified by the given key.
```

Note The function will “touch” the entry and mark it as recently used if the key was found in the cache.

Return This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

Parameters

- `key`: [in] The key for the entry which should be retrieved from the cache.
- `entry`: [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

```
bool insert (key_type const &key, entry_type const &entry)  
Insert a new entry into this cache.
```

Note This function assumes that the entry is not in the cache already. Inserting an already existing entry is considered undefined behavior

Parameters

- `key`: [in] The key for the entry which should be added to the cache.
- `entry`: [in] The entry which should be added to the cache.

```
void insert_nonexist (key_type const &key, entry_type const &entry)
```

```
void update (key_type const &key, entry_type const &entry)  
Update an existing element in this cache.
```

Note The function will “touch” the entry and mark it as recently used if the key was found in the cache.

Note The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Parameters

- `key`: [in] The key for the value which should be updated in the cache.
- `entry`: [in] The entry which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.

```
template<typename F>  
bool update_if (key_type const &key, entry_type const &entry, F &&f)  
Update an existing element in this cache.
```

Note The function will “touch” the entry and mark it as recently used if the key was found in the cache.

Note The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Return This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

Parameters

- *key*: [in] The key for the value which should be updated in the cache.
- *entry*: [in] The value which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.
- *f*: [in] A callable taking two arguments, *k* and the key found in the cache (in that order). If *f* returns true, then the update will continue. If *f* returns false, then the update will not succeed.

`template<typename Func>`
`size_type erase (Func const &ep)`

Remove stored entries from the cache for which the supplied function object returns true.

Return This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry::get_size* functions of the removed entries).

Parameters

- *ep*: [in] This parameter has to be a (unary) function object. It is invoked for each of the entries currently held in the cache. An entry is considered for removal from the cache whenever the value returned from this invocation is *true*.

`size_type erase ()`

Remove all stored entries from the cache.

Return This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry::get_size* functions of the removed entries).

`size_type clear ()`

Clear the cache.

Unconditionally removes all stored entries from the cache.

`statistics_type const &get_statistics () const`

Allow to access the embedded statistics instance.

Return This function returns a reference to the statistics instance embedded inside this cache

`statistics_type &get_statistics ()`

Private Types

```
typedef statistics_type::update_on_exit update_on_exit
```

Private Functions

```
void touch (typename storage_type::iterator it)  
void evict ()
```

Private Members

```
size_type max_size_  
size_type current_size_  
storage_type storage_  
map_type map_  
statistics_type statistics_
```

```
namespace hpx
```

```
namespace util
```

```
namespace cache
```

```
namespace entries
```

```
class entry  
#include <hpx/cache/entries/entry.hpp>
```

Template Parameters

- Value: The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.
- Derived: The (optional) type for which this type is used as a base class.

Public Types

```
typedef Value value_type
```

Public Functions

```
entry ()
```

Any cache entry has to be default constructible.

```
entry (value_type const &val)
```

Construct a new instance of a cache entry holding the given value.

```
bool touch()
```

The function *touch* is called by a cache holding this instance whenever it has been requested (touched).

Note It is possible to change the entry in a way influencing the sort criteria mandated by the UpdatePolicy. In this case the function should return *true* to indicate this to the cache, forcing to reorder the cache entries.

Note This function is part of the CacheEntry concept

Return This function should return *true* if the cache needs to update its internal heap.

Usually this is needed if the entry has been changed by *touch()* in a way influencing the sort order as mandated by the cache's UpdatePolicy

```
bool insert()
```

The function *insert* is called by a cache whenever it is about to be inserted into the cache.

Note This function is part of the CacheEntry concept

Return This function should return *true* if the entry should be added to the cache, otherwise it should return *false*.

```
bool remove()
```

The function *remove* is called by a cache holding this instance whenever it is about to be removed from the cache.

Note This function is part of the CacheEntry concept

Return The return value can be used to avoid removing this instance from the cache. If the value is *true* it is ok to remove the entry, other wise it will stay in the cache.

```
std::size_t get_size() const
```

Return the 'size' of this entry. By default the size of each entry is just one (1), which is sensible if the cache has a limit (capacity) measured in number of entries.

```
value_type &get()
```

Get a reference to the stored data value.

Note This function is part of the CacheEntry concept

```
value_type const &get() const
```

Private Members

```
value_type value_
```

Friends

```
bool operator<(entry const &lhs, entry const &rhs)
```

Forwarding operator< allowing to compare entries instead of the values.

```
namespace hpx
```

```
namespace util
```

```
namespace cache
```

```
namespace entries
```

```
template<typename Value>
class fifo_entry : public hpx::util::cache::entries::entry<Value, fifo_entry<Value>>
#include <hpx/cache/entries/fifo_entry.hpp> The fifo_entry type can be used to store arbitrary values in a cache. Using this type as the cache's entry type makes sure that the least recently inserted entries are discarded from the cache first.
```

Note The fifo_entry conforms to the CacheEntry concept.

Note This type can be used to model a ‘last in first out’ cache policy if it is used with a std::greater as the caches’ UpdatePolicy (instead of the default std::less).

Template Parameters

- **Value**: The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.

Public Functions

fifo_entry()

Any cache entry has to be default constructible.

fifo_entry(Value const &val)

Construct a new instance of a cache entry holding the given value.

bool insert()

The function *insert* is called by a cache whenever it is about to be inserted into the cache.

Note This function is part of the CacheEntry concept

Return This function should return *true* if the entry should be added to the cache, otherwise it should return *false*.

std::chrono::steady_clock::time_point const &get_creation_time() const

Private Types

typedef entry<Value, fifo_entry<Value>> base_type

Private Members

std::chrono::steady_clock::time_point insertion_time_

Friends

```
bool operator<(fifo_entry const &lhs, fifo_entry const &rhs)
    Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been created earlier (FIFO).
```

```
namespace hpx
```

```
namespace util
```

```
namespace cache
```

```
namespace entries
```

```
template<typename Value>
class lfu_entry : public hpx::util::cache::entries::entry<Value, lfu_entry<Value>>
    #include <hpx/cache/entries/lfu_entry.hpp> The lfu_entry type can be used to store arbitrary values in a cache. Using this type as the cache’s entry type makes sure that the least frequently used entries are discarded from the cache first.
```

Note The lfu_entry conforms to the CacheEntry concept.

Note This type can be used to model a ‘most frequently used’ cache policy if it is used with a std::greater as the caches’ UpdatePolicy (instead of the default std::less).

Template Parameters

- **Value**: The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.

Public Functions

```
lfu_entry()
```

Any cache entry has to be default constructible.

```
lfu_entry(Value const &val)
```

Construct a new instance of a cache entry holding the given value.

```
bool touch()
```

The function *touch* is called by a cache holding this instance whenever it has been requested (touched).

In the case of the LFU entry we store the reference count tracking the number of times this entry has been requested. This which will be used to compare the age of an entry during the invocation of the *operator<()*.

Return This function should return true if the cache needs to update its internal heap.

Usually this is needed if the entry has been changed by *touch()* in a way influencing the sort order as mandated by the cache’s UpdatePolicy

```
unsigned long const &get_access_count() const
```

Private Types

```
typedef entry<Value, lfu_entry<Value>> base_type
```

Private Members

```
unsigned long ref_count_
```

Friends

```
bool operator<(lfu_entry const &lhs, lfu_entry const &rhs)
```

Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been accessed less frequently (LFU).

```
namespace hpx
```

```
namespace util
```

```
namespace cache
```

```
namespace entries
```

```
template<typename Value>
```

```
class lru_entry : public hpx::util::cache::entries::entry<Value, lru_entry<Value>>
#include <hpx/cache/entries/lru_entry.hpp> The lru_entry type can be used to store arbitrary values in a cache. Using this type as the cache’s entry type makes sure that the least recently used entries are discarded from the cache first.
```

Note The lru_entry conforms to the CacheEntry concept.

Note This type can be used to model a ‘most recently used’ cache policy if it is used with a std::greater as the caches’ UpdatePolicy (instead of the default std::less).

Template Parameters

- **Value:** The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.

Public Functions

```
lru_entry()
```

Any cache entry has to be default constructible.

```
lru_entry(Value const &val)
```

Construct a new instance of a cache entry holding the given value.

```
bool touch()
```

The function *touch* is called by a cache holding this instance whenever it has been requested (touched).

In the case of the LRU entry we store the time of the last access which will be used to compare the age of an entry during the invocation of the *operator<()*.

Return This function should return true if the cache needs to update its internal heap.
Usually this is needed if the entry has been changed by `touch()` in a way influencing the sort order as mandated by the cache's `UpdatePolicy`

`std::chrono::steady_clock::time_point const &get_access_time() const`
Returns the last access time of the entry.

Private Types

`typedef entry<Value, lru_entry<Value>> base_type`

Private Members

`std::chrono::steady_clock::time_point access_time_`

Friends

`bool operator<(lru_entry const &lhs, lru_entry const &rhs)`

Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been accessed less recently (LRU).

`namespace hpx`

`namespace util`

`namespace cache`

`namespace entries`

`class size_entry`

`#include <hpx/cache/entries/size_entry.hpp>` The `size_entry` type can be used to store values in a cache which have a size associated (such as files, etc.). Using this type as the cache’s entry type makes sure that the entries with the biggest size are discarded from the cache first.

Note The `size_entry` conforms to the `CacheEntry` concept.

Note This type can be used to model a ‘discard smallest first’ cache policy if it is used with a `std::greater` as the caches’ `UpdatePolicy` (instead of the default `std::less`).

Template Parameters

- **Value:** The data type to be stored in a cache. It has to be default constructible, copy constructible and `less_than_comparable`.
- **Derived:** The (optional) type for which this type is used as a base class.

Public Functions

`size_entry()`

Any cache entry has to be default constructible.

`size_entry(Value const &val, std::size_t size)`

Construct a new instance of a cache entry holding the given value.

`std::size_t get_size() const`

Return the ‘size’ of this entry.

Private Types

`typedef detail::size_derived<Value, Derived>::type derived_type`

`typedef entry<Value, derived_type> base_type`

Private Members

`std::size_t size_`

Friends

`bool operator<(size_entry const &lhs, size_entry const &rhs)`

Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has a bigger size.

`namespace hpx`

`namespace util`

`namespace cache`

`namespace policies`

`template<typename Entry>`
`struct always`

Public Functions

`bool operator()(Entry const&)`

`namespace hpx`

`namespace util`

`namespace cache`

```
namespace statistics

class local_full_statistics : public hpx::util::cache::statistics::local_statistics
```

Public Functions

`std::int64_t get_get_entry_count (bool reset)`

The function `get_get_entry_count` returns the number of invocations of the `get_entry()` API function of the cache.

`std::int64_t get_insert_entry_count (bool reset)`

The function `get_insert_entry_count` returns the number of invocations of the `insert_entry()` API function of the cache.

`std::int64_t get_update_entry_count (bool reset)`

The function `get_update_entry_count` returns the number of invocations of the `update_entry()` API function of the cache.

`std::int64_t get_erase_entry_count (bool reset)`

The function `get_erase_entry_count` returns the number of invocations of the `erase()` API function of the cache.

`std::int64_t get_get_entry_time (bool reset)`

The function `get_get_entry_time` returns the overall time spent executing of the `get_entry()` API function of the cache.

`std::int64_t get_insert_entry_time (bool reset)`

The function `get_insert_entry_time` returns the overall time spent executing of the `insert_entry()` API function of the cache.

`std::int64_t get_update_entry_time (bool reset)`

The function `get_update_entry_time` returns the overall time spent executing of the `update_entry()` API function of the cache.

`std::int64_t get_erase_entry_time (bool reset)`

The function `get_erase_entry_time` returns the overall time spent executing of the `erase()` API function of the cache.

Private Functions

`std::int64_t get_and_reset_value (std::int64_t &value, bool reset)`

Private Members

`api_counter_data get_entry_`

`api_counter_data insert_entry_`

`api_counter_data update_entry_`

`api_counter_data erase_entry_`

Friends

```
friend hpx::util::cache::statistics::update_on_exit
struct api_counter_data
```

Public Functions

```
api_counter_data()
```

Public Members

```
std::int64_t count_
std::int64_t time_
struct update_on_exit
#include <local_full_statistics.hpp> Helper class to update timings and counts on function
exit.
```

Public Functions

```
update_on_exit(local_full_statistics &stat, method m)
~update_on_exit()
```

Public Members

```
std::int64_t started_at_
api_counter_data &data_
```

Private Static Functions

```
static api_counter_data &get_api_counter_data(local_full_statistics &stat,
                                              method m)
static std::uint64_t now()

namespace hpx

namespace util

namespace cache

namespace statistics

class local_statistics : public hpx::util::cache::statistics::no_statistics
Subclassed by hpx::util::cache::statistics::local_full_statistics
```

Public Functions

```
local_statistics()

std::size_t get_and_reset (std::size_t &value, bool reset)

std::size_t hits() const

std::size_t misses() const

std::size_t insertions() const

std::size_t evictions() const

std::size_t hits(bool reset)

std::size_t misses(bool reset)

std::size_t insertions(bool reset)

std::size_t evictions(bool reset)

void got_hit()

    The function got_hit will be called by a cache instance whenever a entry got touched.

void got_miss()

    The function got_miss will be called by a cache instance whenever a requested entry has not been found in the cache.

void got_insertion()

    The function got_insertion will be called by a cache instance whenever a new entry has been inserted.

void got_eviction()

    The function got_eviction will be called by a cache instance whenever an entry has been removed from the cache because a new inserted entry let the cache grow beyond its capacity.

void clear()

    Reset all statistics.
```

Private Members

```
std::size_t hits_
std::size_t misses_
std::size_t insertions_
std::size_t evictions_

namespace hpx

namespace util

namespace cache

namespace statistics
```

Enums

```
enum method
    Values:
        method_get_entry = 0
        method_insert_entry = 1
        method_update_entry = 2
        method_erase_entry = 3

class no_statistics
    Subclassed by hpx::util::cache::statistics::local_statistics
```

Public Functions

void got_hit()

The function *got_hit* will be called by a cache instance whenever a entry got touched.

void got_miss()

The function *got_miss* will be called by a cache instance whenever a requested entry has not been found in the cache.

void got_insertion()

The function *got_insertion* will be called by a cache instance whenever a new entry has been inserted.

void got_eviction()

The function *got_eviction* will be called by a cache instance whenever an entry has been removed from the cache because a new inserted entry let the cache grow beyond its capacity.

void clear()

Reset all statistics.

std::int64_t get_get_entry_count (bool)

The function *get_get_entry_count* returns the number of invocations of the *get_entry()* API function of the cache.

std::int64_t get_insert_entry_count (bool)

The function *get_insert_entry_count* returns the number of invocations of the *insert_entry()* API function of the cache.

std::int64_t get_update_entry_count (bool)

The function *get_update_entry_count* returns the number of invocations of the *update_entry()* API function of the cache.

std::int64_t get_erase_entry_count (bool)

The function *get_erase_entry_count* returns the number of invocations of the *erase()* API function of the cache.

std::int64_t get_get_entry_time (bool)

The function *get_get_entry_time* returns the overall time spent executing of the *get_entry()* API function of the cache.

```
std::int64_t get_insert_entry_time(bool)
```

The function *get_insert_entry_time* returns the overall time spent executing of the *insert_entry()* API function of the cache.

```
std::int64_t get_update_entry_time(bool)
```

The function *get_update_entry_time* returns the overall time spent executing of the *update_entry()* API function of the cache.

```
std::int64_t get_erase_entry_time(bool)
```

The function *get_erase_entry_time* returns the overall time spent executing of the *erase()* API function of the cache.

```
struct update_on_exit
```

#include <no_statistics.hpp> Helper class to update timings and counts on function exit.

Public Functions

```
update_on_exit(no_statistics const&, method)
```

command_line_handling_local

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

Enums

```
enum commandline_error_mode
```

Values:

```
return_on_error
```

```
rethrow_on_error
```

```
allow_unregistered
```

```
ignore_aliases = 0x40
```

```
report_missing_config_file = 0x80
```

concepts

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_CONCEPT_REQUIRES_(...)
HPX_CONCEPT_REQUIRES(...)
HPX_CONCEPT_ASSERT(...)
```

Defines

HPX_HAS_MEMBER_XXX_TRAIT_DEF(*MEMBER*)

This macro creates a boolean unary meta-function which result is true if and only if its parameter type has member function with *MEMBER* name (no matter static it is or not). The generated trait ends up in a namespace where the macro itself has been placed.

Defines

HPX_HAS_XXX_TRAIT_DEF(*Name*)

This macro creates a boolean unary meta-function such that for any type X, `has_name<X>::value == true` if and only if X is a class type and has a nested type member `x::name`. The generated trait ends up in a namespace where the macro itself has been placed.

concurrency

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace hpx

namespace util

class barrier

Public Functions

```
barrier(std::size_t number_of_threads)
~barrier()
void wait()
```

Private Types

```
typedef std::mutex mutex_type
```

Private Members

```
const std::size_t number_of_threads_
std::size_t total_
mutex_type mtx_
std::condition_variable cond_
```

Private Static Attributes

```
constexpr std::size_t barrier_flag = static_cast<std::size_t>(1) << (CHAR_BIT * sizeof(std::size_t) - 1)

template<typename Data>
struct cache_aligned_data<Data, std::false_type>
```

Public Functions

```
cache_aligned_data()
cache_aligned_data(Data &&data)
cache_aligned_data(Data const &data)
```

Public Members

```
Data data_
template<typename Data>
struct cache_aligned_data_derived<Data, std::false_type> : public Data
```

Public Functions

```
cache_aligned_data_derived()
cache_aligned_data_derived(Data &&data)
cache_aligned_data_derived(Data const &data)
namespace hpx
```

```
namespace threads
```

Functions

```
constexpr std::size_t get_cache_line_size()
namespace util
```

Typedefs

```
template<typename Data>
using cache_line_data = cache_aligned_data<Data>

template<typename Data, typename NeedsPadding = typename detail::needs_padding<Data>::type>
struct cache_aligned_data
```

Public Functions

```
cache_aligned_data()
cache_aligned_data(Data &&data)
cache_aligned_data(Data const &data)
```

Public Members

```
Data data_
template<>
char cacheline_pad[get_cache_line_padding_size(sizeof(Data))]

template<typename Data>
struct cache_aligned_data<Data, std::false_type>
```

Public Functions

```
cache_aligned_data()
cache_aligned_data(Data &&data)
cache_aligned_data(Data const &data)
```

Public Members

```
Data data_
template<typename Data, typename NeedsPadding = typename detail::needs_padding<Data>::type>
struct cache_aligned_data_derived : public Data
```

Public Functions

```
cache_aligned_data_derived()
cache_aligned_data_derived(Data &&data)
cache_aligned_data_derived(Data const &data)
```

Public Members

```
template<>
char cacheline_pad[get_cache_line_padding_size(sizeof(Data))]

template<typename Data>
struct cache_aligned_data_derived<Data, std::false_type> : public Data
```

Public Functions

```
cache_aligned_data_derived()
cache_aligned_data_derived(Data &&data)
cache_aligned_data_derived(Data const &data)
```

Defines

```
MOODYCAMEL_THREADLOCAL
MOODYCAMEL_EXCEPTIONS_ENABLED
MOODYCAMEL_TRY
MOODYCAMEL_CATCH(...)
MOODYCAMEL_RETHROW
MOODYCAMEL_THROW(expr)
MOODYCAMEL_NOEXCEPT
MOODYCAMEL_NOEXCEPTCTOR(type, valueType, expr)
MOODYCAMEL_NOEXCEPT_ASSIGN(type, valueType, expr)
MOODYCAMEL_DELETE_FUNCTION
namespace hpx

namespace concurrency
```

Functions

```
template<typename T, typename Traits>
void swap(typename ConcurrentQueue<T, Traits>::ImplicitProducerKVP &a, typename ConcurrentQueue<T, Traits>::ImplicitProducerKVP &b)

template<typename T, typename Traits>
void swap(ConcurrentQueue<T, Traits> &a, ConcurrentQueue<T, Traits> &b)

void swap(ProducerToken &a, ProducerToken &b)
void swap(ConsumerToken &a, ConsumerToken &b)

template<typename T, typename Traits = ConcurrentQueueDefaultTraits>
class ConcurrentQueue
```

Public Types

```
typedef ::hpx::concurrency::ProducerToken producer_token_t
typedef ::hpx::concurrency::ConsumerToken consumer_token_t
typedef Traits::index_t index_t
typedef Traits::size_t size_t
```

Public Functions

```
ConcurrentQueue (size_t capacity = 6 * BLOCK_SIZE)
ConcurrentQueue (size_t minCapacity, size_t maxExplicitProducers, size_t maxImplicitProducers)
~ConcurrentQueue ()
ConcurrentQueue (ConcurrentQueue const&)
ConcurrentQueue &operator= (ConcurrentQueue const&)
ConcurrentQueue (ConcurrentQueue &&other)
ConcurrentQueue &operator= (ConcurrentQueue &&other)
void swap (ConcurrentQueue &other)
bool enqueue (T const &item)
bool enqueue (T &&item)
bool enqueue (producer_token_t const &token, T const &item)
bool enqueue (producer_token_t const &token, T &&item)
template<typename It>
bool enqueue_bulk (It itemFirst, size_t count)
template<typename It>
bool enqueue_bulk (producer_token_t const &token, It itemFirst, size_t count)
bool try_enqueue (T const &item)
bool try_enqueue (T &&item)
bool try_enqueue (producer_token_t const &token, T const &item)
bool try_enqueue (producer_token_t const &token, T &&item)
template<typename It>
bool try_enqueue_bulk (It itemFirst, size_t count)
template<typename It>
bool try_enqueue_bulk (producer_token_t const &token, It itemFirst, size_t count)
template<typename U>
bool try_dequeue (U &item)
```

```
template<typename U>
bool try_dequeue_non_interleaved(U &item)

template<typename U>
bool try_dequeue (consumer_token_t &token, U &item)

template<typename It>
size_t try_dequeue_bulk (It itemFirst, size_t max)

template<typename It>
size_t try_dequeue_bulk (consumer_token_t &token, It itemFirst, size_t max)

template<typename U>
bool try_dequeue_from_producer (producer_token_t const &producer, U &item)

template<typename It>
size_t try_dequeue_bulk_from_producer (producer_token_t const &producer, It
                                         itemFirst, size_t max)

size_t size_approx () const
```

Public Static Functions

```
static bool is_lock_free ()
```

Public Static Attributes

```
const size_t BLOCK_SIZE = static_cast<size_t>(Traits::BLOCK_SIZE)
const size_t EXPLICIT_BLOCK_EMPTY_COUNTER_THRESHOLD = static_cast<size_t>(Traits::EXPLICIT_BLOCK_EMPTY_COUNTER_THRESHOLD)
const size_t EXPLICIT_INITIAL_INDEX_SIZE = static_cast<size_t>(Traits::EXPLICIT_INITIAL_INDEX_SIZE)
const size_t IMPLICIT_INITIAL_INDEX_SIZE = static_cast<size_t>(Traits::IMPLICIT_INITIAL_INDEX_SIZE)
const size_t INITIAL_IMPLICIT_PRODUCER_HASH_SIZE = static_cast<size_t>(Traits::INITIAL_IMPLICIT_PRODUCER_HASH_SIZE)
const std::uint32_t EXPLICIT_CONSUMER_CONSUMPTION_QUOTA_BEFORE_ROTATE = static_cast<std::uint32_t>(Traits::EXPLICIT_CONSUMER_CONSUMPTION_QUOTA_BEFORE_ROTATE)
const size_t hpx::concurrency::ConcurrentQueue::MAX_SUBQUEUE_SIZE = (details::concurrent_queue::MAX_SUBQUEUE_SIZE)
```

Private Types

```
enum AllocationMode
Values:
    CanAlloc
    CannotAlloc

enum InnerQueueContext
Values:
    implicit_context = 0
    explicit_context = 1
```

Private Functions

```

ConcurrentQueue &swap_internal (ConcurrentQueue &other)

template<AllocationMode canAlloc, typename U>
bool inner_enqueue (producer_token_t const &token, U &&element)

template<AllocationMode canAlloc, typename U>
bool inner_enqueue (U &&element)

template<AllocationMode canAlloc, typename It>
bool inner_enqueue_bulk (producer_token_t const &token, It itemFirst, size_t count)

template<AllocationMode canAlloc, typename It>
bool inner_enqueue_bulk (It itemFirst, size_t count)

bool update_current_producer_after_rotation (consumer_token_t &token)

void populate_initial_block_list (size_t blockCount)

Block *try_get_block_from_initial_pool ()

void add_block_to_free_list (Block *block)

void add_blocks_to_free_list (Block *block)

Block *try_get_block_from_free_list ()

template<AllocationMode canAlloc>
Block *requisition_block ()

ProducerBase *recycle_or_create_producer (bool isExplicit)

ProducerBase *recycle_or_create_producer (bool isExplicit, bool &recycled)

ProducerBase *add_producer (ProducerBase *producer)

void reown_producers ()

void populate_initial_implicit_producer_hash ()

void swap_implicit_producer_hashes (ConcurrentQueue &other)

ImplicitProducer *get_or_add_implicit_producer ()

```

Private Members

```

std::atomic<ProducerBase*> producerListTail
std::atomic<std::uint32_t> producerCount
std::atomic<size_t> initialBlockPoolIndex
Block *initialBlockPool
size_t initialBlockPoolSize
FreeList<Block> freeList
std::atomic<ImplicitProducerHash*> implicitProducerHash

```

```
std::atomic<size_t> implicitProducerHashCount
ImplicitProducerHash initialImplicitProducerHash
std::array<ImplicitProducerKVP, INITIAL_IMPLICIT_PRODUCER_HASH_SIZE> initialImplicitProducerHashes
std::atomic_flag implicitProducerHashResizeInProgress
std::atomic<std::uint32_t> nextExplicitConsumerId
std::atomic<std::uint32_t> globalExplicitConsumerOffset
```

Private Static Functions

```
template<typename U>
static U *create_array(size_t count)

template<typename U>
static void destroy_array(U *p, size_t count)

template<typename U>
static U *create()

template<typename U, typename A1>
static U *create(A1 &&a1)

template<typename U>
static void destroy(U *p)
```

Friends

```
friend hpx::concurrency::ProducerToken
friend hpx::concurrency::ConsumerToken
friend hpx::concurrency::ExplicitProducer
friend hpx::concurrency::ImplicitProducer
friend hpx::concurrency::ConcurrentQueueTests

template<typename XT, typename XTraits>
void swap(typename ConcurrentQueue<XT, XTraits>::ImplicitProducerKVP&, typename
          ConcurrentQueue<XT, XTraits>::ImplicitProducerKVP&)

struct Block
```

Public Functions

```
template<>
Block()

template<InnerQueueContext context>
bool is_empty() const

template<InnerQueueContext context>
bool set_empty(index_t i)

template<InnerQueueContext context>
```

```

bool set_many_empty(index_t i, size_t count)
template<InnerQueueContext context>
void set_all_empty()

template<InnerQueueContext context>
void reset_empty()

template<>
T *operator[](index_t idx)
template<>
T const *operator[](index_t idx) const

```

Public Members

```

template<>
char elements[sizeof(T) * BLOCK_SIZE]
template<>
details::max_align_t dummy
template<>
Block *next
template<>
std::atomic<size_t> elementsCompletelyDequeued
std::atomic<bool> hpx::concurrency::ConcurrentQueue< T, Traits >::Block::empty
template<>
std::atomic<std::uint32_t> freeListRefs
template<>
std::atomic<Block*> freeListNext
template<>
std::atomic<bool> shouldBeOnFreeList
template<>
bool dynamicallyAllocated

```

Private Members

```

template<>
union hpx::concurrency::ConcurrentQueue::Block::[anonymous] [anonymous]
struct ExplicitProducer : public hpx::concurrency::ConcurrentQueue<T, Traits>::ProducerBase

```

Public Functions

```
template<>
ExplicitProducer (ConcurrentQueue *parent)

template<>
~ExplicitProducer ()

template<AllocationMode allocMode, typename U>
bool enqueue (U &&element)

template<typename U>
bool dequeue (U &element)

template<AllocationMode allocMode, typename It>
bool enqueue_bulk (It itemFirst, size_t count)

template<typename It>
size_t dequeue_bulk (It &itemFirst, size_t max)
```

Private Functions

```
template<>
bool new_block_index (size_t numberOffilledSlotsToExpose)
```

Private Members

```
template<>
std::atomic<BlockIndexHeader*> blockIndex

template<>
size_t pr_blockIndexSlotsUsed

template<>
size_t pr_blockIndexSize

template<>
size_t pr_blockIndexFront

template<>
BlockIndexEntry *pr_blockIndexEntries

template<>
void *pr_blockIndexRaw

struct BlockIndexEntry
```

Public Members

```
template<>
index_t base

template<>
Block *block

struct BlockIndexHeader
```

Public Members

```
template<>
size_t size

template<>
std::atomic<size_t> front

template<>
BlockIndexEntry *entries

template<>
void *prev

template<typename N>
struct FreeList
```

Public Functions

```
template<>
FreeList ()

template<>
FreeList (FreeList &&other)

template<>
void swap (FreeList &other)

template<>
FreeList (FreeList const&)

template<>
FreeList &operator= (FreeList const&)

template<>
void add (N *node)

template<>
N *try_get ()

template<>
N *head_unsafe () const
```

Private Functions

```
template<>
void add_knowing_refcount_is_zero (N *node)
```

Private Members

```
template<>
std::atomic<N*> freeListHead
```

Private Static Attributes

```
template<>
const std::uint32_t REFS_MASK = 0x7FFFFFFF
template<>
const std::uint32_t SHOULD_BE_ON_FREELIST = 0x80000000
template<typename N>
struct FreeListNode
```

Public Functions

```
template<>
FreeListNode ()
```

Public Members

```
template<>
std::atomic<std::uint32_t> freeListRefs
template<>
std::atomic<N*> freeListNext
struct ImplicitProducer : public hpx::concurrency::ConcurrentQueue<T, Traits>::ProducerBase
```

Public Functions

```
template<>
ImplicitProducer (ConcurrentQueue *parent)
template<>
~ImplicitProducer ()
template<AllocationMode allocMode, typename U>
bool enqueue (U &&element)
template<typename U>
bool dequeue (U &element)
template<AllocationMode allocMode, typename It>
bool enqueue_bulk (It itemFirst, size_t count)
```

```
template<typename It>
size_t dequeue_bulk (It &itemFirst, size_t max)
```

Private Functions

```
template<AllocationMode allocMode>
bool insert_block_index_entry (BlockIndexEntry *&idxEntry, index_t blockStartIndex)

template<>
void rewind_block_index_tail ()

template<>
BlockIndexEntry *get_block_index_entry_for_index (index_t index) const

template<>
size_t get_block_index_index_for_index (index_t index, BlockIndexHeader *&localBlockIndex) const

template<>
bool new_block_index ()
```

Private Members

```
template<>
size_t nextBlockIndexCapacity

template<>
std::atomic<BlockIndexHeader*> blockIndex
```

Private Static Attributes

```
template<>
const index_t INVALID_BLOCK_BASE = 1

struct BlockIndexEntry
```

Public Members

```
template<>
std::atomic<index_t> key

template<>
std::atomic<Block*> value

struct BlockIndexHeader
```

Public Members

```
template<>
size_t capacity

template<>
std::atomic<size_t> tail

template<>
BlockIndexEntry *entries

template<>
BlockIndexEntry **index

template<>
BlockIndexHeader *prev

struct ImplicitProducerHash
```

Public Members

```
template<>
size_t capacity

template<>
ImplicitProducerKVP *entries

template<>
ImplicitProducerHash *prev

struct ImplicitProducerKVP
```

Public Functions

```
template<>
ImplicitProducerKVP ()

template<>
ImplicitProducerKVP (ImplicitProducerKVP &&other)

template<>
ImplicitProducerKVP &operator= (ImplicitProducerKVP &&other)

template<>
void swap (ImplicitProducerKVP &other)
```

Public Members

```
template<>
std::atomic<details::thread_id_t> key

template<>
ImplicitProducer *value

struct ProducerBase : public hpx::concurrency::details::ConcurrentQueueProducerTypelessBase
```

Public Functions

```
template<>
ProducerBase (ConcurrentQueue *parent_, bool isExplicit_)

template<>
virtual ~ProducerBase ()

template<typename U>
bool dequeue (U &element)

template<typename It>
size_t dequeue_bulk (It &itemFirst, size_t max)

template<>
ProducerBase *next_prod () const

template<>
size_t size_approx () const

template<>
index_t getTail () const
```

Public Members

```
template<>
bool isExplicit

template<>
ConcurrentQueue *parent
```

Protected Attributes

```
template<>
std::atomic<index_t> tailIndex

template<>
std::atomic<index_t> headIndex

template<>
std::atomic<index_t> dequeueOptimisticCount

template<>
std::atomic<index_t> dequeueOvercommit

template<>
Block *tailBlock

struct ConcurrentQueueDefaultTraits
```

Public Types

```
typedef std::size_t size_t
typedef std::size_t index_t
```

Public Static Functions

```
static void *malloc (size_t size)
static void free (void *ptr)
```

Public Static Attributes

```
const size_t BLOCK_SIZE = 32
const size_t EXPLICIT_BLOCK_EMPTY_COUNTER_THRESHOLD = 32
const size_t EXPLICIT_INITIAL_INDEX_SIZE = 32
const size_t IMPLICIT_INITIAL_INDEX_SIZE = 32
const size_t INITIAL_IMPLICIT_PRODUCER_HASH_SIZE = 32
const std::uint32_t EXPLICIT_CONSUMER_CONSUMPTION_QUOTA_BEFORE_ROTATE = 256
const size_t MAX_SUBQUEUE_SIZE = details::const_numeric_max<size_t>::value

struct ConsumerToken
```

Public Functions

```
template<typename T, typename Traits>
ConsumerToken (ConcurrentQueue<T, Traits> &q)

template<typename T, typename Traits>
ConsumerToken (BlockingConcurrentQueue<T, Traits> &q)

ConsumerToken (ConsumerToken &&other)

ConsumerToken &operator= (ConsumerToken &&other)

void swap (ConsumerToken &other)

ConsumerToken (ConsumerToken const&)

ConsumerToken &operator= (ConsumerToken const&)
```

Private Members

```
std::uint32_t initialOffset
std::uint32_t lastKnownGlobalOffset
std::uint32_t itemsConsumedFromCurrent
details::ConcurrentQueueProducerTypelessBase *currentProducer
details::ConcurrentQueueProducerTypelessBase *desiredProducer
```

Friends

```
friend hpx::concurrency::ConcurrentQueue
friend hpx::concurrency::ConcurrentQueueTests
struct ProducerToken
```

Public Functions

```
template<typename T, typename Traits>
ProducerToken (ConcurrentQueue<T, Traits> &queue)

template<typename T, typename Traits>
ProducerToken (BlockingConcurrentQueue<T, Traits> &queue)

ProducerToken (ProducerToken &&other)

ProducerToken &operator= (ProducerToken &&other)

void swap (ProducerToken &other)

bool valid () const

~ProducerToken ()

ProducerToken (ProducerToken const&)

ProducerToken &operator= (ProducerToken const&)
```

Protected Attributes

```
details::ConcurrentQueueProducerTypelessBase *producer
```

Friends

```
friend hpx::concurrency::ConcurrentQueue
friend hpx::concurrency::ConcurrentQueueTests
namespace details
```

TypeDefs

```
typedef std::uintptr_t thread_id_t  
typedef std::max_align_t std_max_align_t
```

Functions

```
static thread_id_t thread_id()  
  
static bool() hpx::concurrency::details::likely(bool x)  
static bool() hpx::concurrency::details::unlikely(bool x)  
  
static size_t hash_thread_id(thread_id_t id)  
  
template<typename T>  
static bool circular_less_than(T a, T b)  
  
template<typename U>  
static char *align_for(char *ptr)  
  
template<typename T>  
static T ceil_to_pow_2(T x)  
  
template<typename T>  
static void swap_relaxed(std::atomic<T> &left, std::atomic<T> &right)  
  
template<typename T>  
static T const &nomove(T const &x)  
  
template<typename It>  
static auto deref_noexcept(It &it)
```

Variables

```
const thread_id_t invalid_thread_id = 0  
const thread_id_t invalid_thread_id2 = 1  
  
template<bool use32>  
struct _hash_32_or_64
```

Public Static Functions

```
static std::uint32_t hash(std::uint32_t h)  
  
template<>  
struct _hash_32_or_64<1>
```

Public Static Functions

```
static std::uint64_t hash (std::uint64_t h)

struct ConcurrentQueueProducerTypelessBase
```

Public Functions

```
ConcurrentQueueProducerTypelessBase ()
```

Public Members

```
ConcurrentQueueProducerTypelessBase *next
std::atomic<bool> inactive
ProducerToken *token

template<typename T>
struct const_numeric_max
```

Public Static Attributes

```
const T hpx::concurrency::details::const_numeric_max::value= std::numeric_lim
union max_align_t
```

Public Members

```
std_max_align_t x
long long y
void *z

template<bool Enable>
struct nomove_if
```

Public Static Functions

```
template<typename T>
static T const &eval (T const &x)

template<>
struct nomove_if<false>
```

Public Static Functions

```
template<typename U>
static auto eval (U &&x)

template<>
struct static_is_lock_free<bool>
```

Public Types

```
enum [anonymous]
Values:
value = ATOMIC_BOOL_LOCK_FREE
```

```
template<typename U>
struct static_is_lock_free<U*>
```

Public Types

```
enum [anonymous]
Values:
value = ATOMIC_POINTER_LOCK_FREE
```

```
template<typename T>
struct static_is_lock_free_num
```

Public Types

```
enum [anonymous]
Values:
value = 0
```

```
template<>
struct static_is_lock_free_num<int>
```

Public Types

```
enum [anonymous]
Values:
value = ATOMIC_INT_LOCK_FREE
```

```
template<>
struct static_is_lock_free_num<long>
```

Public Types

```
enum [anonymous]
Values:
    value = ATOMIC_LONG_LOCK_FREE

template<>
struct static_is_lock_free_num<long long>
```

Public Types

```
enum [anonymous]
Values:
    value = ATOMIC_LLONG_LOCK_FREE

template<>
struct static_is_lock_free_num<short>
```

Public Types

```
enum [anonymous]
Values:
    value = ATOMIC_SHORT_LOCK_FREE

template<>
struct static_is_lock_free_num<signed char>
```

Public Types

```
enum [anonymous]
Values:
    value = ATOMIC_CHAR_LOCK_FREE

template<typename thread_id_t>
struct thread_id_converter
```

Public Types

```
typedef thread_id_t thread_id_numeric_size_t
typedef thread_id_t thread_id_hash_t
```

Public Static Functions

```
static thread_id_hash_t prehash(thread_id_t const &x)

namespace boost

namespace lockfree
```

Enums

```
enum deque_status_type
```

Values:

```
stable
```

```
rpush
```

```
lpush
```

```
template<typename T, typename freelist_t = caching_freelist_t, typename Alloc = std::allocator<T>>
struct deque
```

Public Types

```
template<>
```

```
using node = deque_node<T>
```

```
template<>
```

```
using node_pointer = typename node::pointer
```

```
template<>
```

```
using atomic_node_pointer = typename node::atomic_pointer
```

```
template<>
```

```
using tag_t = typename node::tag_t
```

```
template<>
```

```
using anchor = deque_anchor<T>
```

```
template<>
```

```
using anchor_pair = typename anchor::pair
```

```
template<>
```

```
using atomic_anchor_pair = typename anchor::atomic_pair
```

```
template<>
```

```
using node_allocator = typename std::allocator_traits<Alloc>::template rebind_alloc<node>
```

```
template<>
```

```
using pool = typename std::conditional<std::is_same<freelist_t, caching_freelist_t>::value, caching_freelist<no
```

Public Functions

```
HPX_NON_COPYABLE (deque)
deque (std::size_t initial_nodes = 128)
~deque ()
bool empty () const
bool is_lock_free () const
bool push_left (T data)
bool push_right (T data)
bool pop_left (T &r)
bool pop_left (T *r)
bool pop_right (T &r)
bool pop_right (T *r)
```

Private Functions

```
node *alloc_node (node *lptr, node *rptr, T const &v, tag_t ltag = 0, tag_t rtag = 0)
node *alloc_node (node *lptr, node *rptr, T &&v, tag_t ltag = 0, tag_t rtag = 0)
void dealloc_node (node *n)
void stabilize_left (anchor_pair &lrs)
void stabilize_right (anchor_pair &lrs)
void stabilize (anchor_pair &lrs)
```

Private Members

```
anchor anchor_
pool pool_
template<*>
char padding[padding_size]
```

Private Static Attributes

```
constexpr std::size_t padding_size = BOOST_LOCKFREE_CACHELINE_BYTES - sizeof(anchor)
template<typename T>
struct deque_anchor
```

Public Types

```
template<>
using node = deque_node<T>

template<>
using node_pointer = typename node::pointer

template<>
using atomic_node_pointer = typename node::atomic_pointer

template<>
using tag_t = typename node::tag_t

template<>
using anchor = deque_anchor<T>

template<>
using pair = tagged_ptr_pair<node, node>

template<>
using atomic_pair = std::atomic<pair>
```

Public Functions

```
deque_anchor ()

deque_anchor (deque_anchor const &p)

deque_anchor (pair const &p)

deque_anchor (node *lptr, node *rptr, tag_t status = stable, tag_t tag = 0)

pair lrs (std::memory_order mo = std::memory_order_acquire) volatile const

node *left (std::memory_order mo = std::memory_order_acquire) volatile const

node *right (std::memory_order mo = std::memory_order_acquire) volatile const

tag_t status (std::memory_order mo = std::memory_order_acquire) volatile const

tag_t tag (std::memory_order mo = std::memory_order_acquire) volatile const

bool cas (deque_anchor &expected, deque_anchor const &desired, std::memory_order mo =
    std::memory_order_acq_rel) volatile

bool cas (pair &expected, deque_anchor const &desired, std::memory_order mo =
    std::memory_order_acq_rel) volatile

bool cas (deque_anchor &expected, pair const &desired, std::memory_order mo =
    std::memory_order_acq_rel) volatile

bool cas (pair &expected, pair const &desired, std::memory_order mo =
    std::memory_order_acq_rel) volatile

bool operator== (volatile deque_anchor const &rhs) const

bool operator!= (volatile deque_anchor const &rhs) const

bool operator== (volatile pair const &rhs) const
```

```
bool operator!=(volatile pair const &rhs) const
bool is_lock_free() const
```

Private Members

```
atomic_pair pair_
```

```
template<typename T>
struct deque_node
```

Public Types

```
typedef detail::tagged_ptr<deque_node> pointer
typedef std::atomic<pointer> atomic_pointer
typedef pointer::tag_t tag_t
```

Public Functions

```
deque_node()
deque_node(deque_node const &p)
deque_node(deque_node *lptr, deque_node *rptr, T const &v, tag_t ltag = 0, tag_t rtag = 0)
deque_node(deque_node *lptr, deque_node *rptr, T &&v, tag_t ltag = 0, tag_t rtag = 0)
```

Public Members

```
atomic_pointer left
```

```
atomic_pointer right
```

```
T data
```

```
namespace hpx
```

```
namespace util
```

```
struct spinlock
#include <spinlock.hpp> Lockable spinlock class.
```

Public Functions

```
HPX_NON_COPYABLE (spinlock)
spinlock (char const* = nullptr)
~spinlock ()
void lock ()
bool try_lock ()
void unlock ()
```

Private Members

```
hpx::util::detail::spinlock m
namespace hpx

namespace util

template<typename Tag, std::size_t N = HPX_HAVE_SPINLOCK_POOL_NUM>
class spinlock_pool
```

Public Static Functions

```
static detail::spinlock &spinlock_for (void const *pv)
```

Private Static Attributes

```
cache_aligned_data<detail::spinlock> pool_
```

config

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_INITIAL_IP_PORT

This is the default ip/port number used by the parcel subsystem.

HPX_CONNECTING_IP_PORT

HPX_INITIAL_IP_ADDRESS

HPX_RUNTIME_INSTANCE_LIMIT

This defines the maximum number of possible runtime instances in one executable

HPX_PARCEL_BOOTSTRAP

This defines the type of the parcelport to be used during application bootstrap. This value can be changed at runtime by the configuration parameter:

```
hpx.parcel.bootstrap = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_BOOTSTRAP).

HPX_PARCEL_MAX_CONNECTIONS

This defines the number of outgoing (parcel-) connections kept alive (to all other localities). This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_connections = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_MAX_CONNECTIONS).

HPX_PARCEL_IPC_DATA_BUFFER_CACHE_SIZE

This defines the number of outgoing ipc (parcel-) connections kept alive (to each of the other localities on the same node). This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.ipc.data_buffer_cache_size = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_IPC_DATA_BUFFER_CACHE_SIZE).

HPX_PARCEL_MPI_MAX_REQUESTS

This defines the number of MPI requests in flight. This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.mpi.max_requests = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_MPI_MAX_REQUESTS).

HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY

This defines the number of outgoing (parcel-) connections kept alive (to each of the other localities). This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_connections_per_locality = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY).

HPX_PARCEL_MAX_MESSAGE_SIZE

This defines the maximally allowed message size for messages transferred between localities. This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_message_size = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_MAX_MESSAGE_SIZE).

HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE

This defines the maximally allowed outbound message size for coalescing messages transferred between localities. This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_outbound_message_size = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE).

HPX_PARCEL_SERIALIZATION_OVERHEAD**HPX_AGAS_LOCAL_CACHE_SIZE**

This defines the number of AGAS address translations kept in the local cache. This is just the initial size which may be adjusted depending on the load of the system (not implemented yet), etc. It must be a minimum of 3 for AGAS v3 bootstrapping.

This value can be changes at runtime by setting the configuration parameter:

```
hpx.agas.local_cache_size = ...
```

(or by setting the corresponding environment variable HPX_AGAS_LOCAL_CACHE_SIZE)

HPX_INITIAL_AGAS_MAX_PENDING_REFCNT_REQUESTS

HPX_GLOBALCREDIT_INITIAL

This defines the initial global reference count associated with any created object.

HPX_NUM_IO_POOL_SIZE

This defines the default number of OS-threads created for the different internal thread pools

HPX_NUM_PARCEL_POOL_SIZE

HPX_NUM_TIMER_POOL_SIZE

HPX_SPINLOCK_DEADLOCK_DETECTION_LIMIT

By default, enable minimal thread deadlock detection in debug builds only.

HPX_COROUTINE_NUM_HEAPS

This defines the default number of coroutine heaps.

HPX_HAVE_THREAD_BACKTRACE_DEPTH

By default we capture only 20 levels of stack back trace on suspension.

By default we do not maintain stack back-traces on suspension. This is a pure debugging aid to be able to see in the debugger where a suspended thread got stuck.

HPX_MAX_NETWORK_RETRIES

HPX_NETWORK_RETRIES_SLEEP

HPX_INI_PATH_DELIMITER

HPX_PATH_DELIMITERS

HPX_SHARED_LIB_EXTENSION

HPX_EXECUTABLE_EXTENSION

HPX_MAKE_DLL_STRING (n)

HPX_MANGLE_NAME (n)

HPX_MANGLE_STRING (n)

HPX_COMPONENT_NAME_DEFAULT

HPX_COMPONENT_NAME

HPX_COMPONENT_STRING

HPX_PLUGIN_COMPONENT_PREFIX

HPX_PLUGIN_NAME_DEFAULT

HPX_PLUGIN_NAME

HPX_PLUGIN_STRING

HPX_PLUGIN_PLUGIN_PREFIX

HPX_IDLE_LOOP_COUNT_MAX

HPX_BUSY_LOOP_COUNT_MAX

HPX_THREAD_QUEUE_MAX_THREAD_COUNT

HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_PENDING

HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_STAGED

```
HPX_THREAD_QUEUE_MIN_ADD_NEW_COUNT  
HPX_THREAD_QUEUE_MAX_ADD_NEW_COUNT  
HPX_THREAD_QUEUE_MIN_DELETE_COUNT  
HPX_THREAD_QUEUE_MAX_DELETE_COUNT  
HPX_THREAD_QUEUE_MAX_TERMINATED_THREADS  
HPX_THREAD_QUEUE_INIT_THREADS_COUNT  
HPX_IDLE_BACKOFF_TIME_MAX  
HPX_WRAPPER_HEAP_STEP  
HPX_INITIAL_GID_RANGE  
HPX_CONTINUATION_MAX_RECURSION_DEPTH
```

Defines

HPX_NOLINK

Function attribute to tell compiler not to link the function.

HPX_NORETURN

Function attribute to tell compiler that the function does not return.

HPX_DEPRECATED (x)

Marks an entity as deprecated. The argument *x* specifies a custom message that is included in the compiler warning. For more details see <>__.

HPX_FALLTHROUGH

Indicates that the fall through from the previous case label is intentional and should not be diagnosed by a compiler that warns on fallthrough. For more details see <>__.

HPX_NODISCARD

If a function declared nodiscard or a function returning an enumeration or class declared nodiscard by value is called from a discarded-value expression other than a cast to void, the compiler is encouraged to issue a warning. For more details see __.

HPX_NO_UNIQUE_ADDRESS

Indicates that this data member need not have an address distinct from all other non-static data members of its class. For more details see __.

Defines

HPX_LIKELY (expr)

Hint at the compiler that *expr* is likely to be true.

HPX_UNLIKELY (expr)

Hint at the compiler that *expr* is likely to be false.

Defines

HPX_COMPILER_FENCE

Generates assembly that serves as a fence to the compiler CPU to disable optimization. Usually implemented in the form of a memory barrier.

HPX_SMT_PAUSE

Generates assembly that executes a “pause” instruction. Useful in spinning loops.

Defines

HPX_GCC_VERSION

Returns the GCC version HPX is compiled with. Only set if compiled with GCC.

HPX_CLANG_VERSION

Returns the Clang version HPX is compiled with. Only set if compiled with Clang.

HPX_INTEL_VERSION

Returns the Intel Compiler version HPX is compiled with. Only set if compiled with the Intel Compiler.

HPX_MSVC

This macro is set if the compilation is with MSVC.

HPX_MINGW

This macro is set if the compilation is with Mingw.

HPX_WINDOWS

This macro is set if the compilation is for Windows.

HPX_NATIVE_MIC

This macro is set if the compilation is for Intel Knights Landing.

Defines

HPX_HOST_DEVICE_INLINE_CONSTEXPR_VARIABLE

This macro evaluates to `inline constexpr` for host code and.

Defines

HPX_DEBUG

Defined if HPX is compiled in debug mode.

HPX_BUILD_TYPE

Evaluates to `debug` if compiled in debug mode, `release` otherwise.

Defines

HPX_HAVE_DEPRECATED_WARNINGS_V1_4

HPX_DEPRECATED_V1_4 (*x*)

HPX_HAVE_DEPRECATED_WARNINGS_V1_5

HPX_DEPRECATED_V1_5 (*x*)

HPX_HAVE_DEPRECATED_WARNINGS_V1_6

HPX_DEPRECATED_V1_6 (*x*)

`HPX_HAVE_DEPRECATED_WARNINGS_V1_7`

`HPX_DEPRECATED_V1_7 (x)`

`HPX_HAVE_DEPRECATED_WARNINGS_V1_8`

`HPX_DEPRECATED_V1_8 (x)`

`HPX_DEPRECATED_V (major, minor, x)`

Defines

`HPX_NON_COPYABLE (cls)`

Marks a class as non-copyable and non-movable.

Defines

`HPX_EXPORT`

Marks a class or function to be exported from HPX or imported if it is consumed.

Defines

`HPX_FORCEINLINE`

Marks a function to be forced inline.

Defines

`HPX_FORWARD (T, ...)`

Defines

`HPX_CXX20_CAPTURE_THIS (...)`

Defines

`HPX_SUPER_PURE`

`HPX_PURE`

`HPX_HOT`

`HPX_COLD`

Defines

`HPX_MOVE (...)`

Defines

```
HPX_THREADS_STACK_OVERHEAD  
HPX_SMALL_STACK_SIZE  
HPX_MEDIUM_STACK_SIZE  
HPX_LARGE_STACK_SIZE  
HPX_HUGE_STACK_SIZE
```

Defines

```
HPX_WEAK_SYMBOL
```

config_registry

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_CONFIG_REGISTRY_EXPORT  
namespace hpx
```

```
namespace config_registry
```

Functions

```
HPX_CONFIG_REGISTRY_EXPORT std::vector<module_config> const& hpx::config_registry::  
HPX_CONFIG_REGISTRY_EXPORT void hpx::config_registry::add_module_config(module_config  
struct add_module_config_helper
```

Public Functions

```
add_module_config_helper(module_config const &config)  
struct module_config
```

Public Members

```
std::string module_name  
std::vector<std::string> config_entries
```

coroutines

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
  namespace threads
```

```
    namespace coroutines
```

```
      class coroutine
```

Public Types

```
      using impl_type = detail::coroutine_impl
      using thread_id_type = impl_type::thread_id_type
      using result_type = impl_type::result_type
      using arg_type = impl_type::arg_type
      using functor_type = util::unique_function_nonser<result_type (arg_type)>
```

Public Functions

```
  coroutine (functor_type &&f, thread_id_type id, std::ptrdiff_t stack_size = detail::default_stack_size)
  coroutine (coroutine const &src)
  coroutine &operator= (coroutine const &src)
  coroutine (coroutine &&src)
  coroutine &operator= (coroutine &&src)
  thread_id_type get_thread_id () const
  std::size_t get_thread_data () const
  std::size_t set_thread_data (std::size_t data)
  void init ()
  void rebind (functor_type &&f, thread_id_type id)
  result_type operator() (arg_type arg = arg_type ())
  bool is_ready () const
  std::ptrdiff_t get_available_stack_space ()
  impl_type *impl ()
```

Private Members

```
impl_type impl_

namespace hpx

namespace threads

namespace coroutines

class stackless_coroutine
```

Public Types

```
using thread_id_type = hpx::threads::thread_id
using result_type = std::pair<thread_schedule_state, thread_id_type>
using arg_type = thread_restart_state
using functor_type = util::unique_function_nonsig<result_type (arg_type)>
```

Public Functions

```
stackless_coroutine(functor_type &&f, thread_id_type id, std::ptrdiff_t = default_stack_size)
~stackless_coroutine()

stackless_coroutine(stackless_coroutine const &src)

stackless_coroutine &operator=(stackless_coroutine const &src)

stackless_coroutine(stackless_coroutine &&src)

stackless_coroutine &operator=(stackless_coroutine &&src)

thread_id_type get_thread_id() const
std::size_t get_thread_data() const
std::size_t set_thread_data(std::size_t data)
void rebind(functor_type &&f, thread_id_type id)
void reset_tss()
void reset()

stackless_coroutine::result_type operator() (arg_type arg = arg_type())
operator bool() const
bool is_ready() const
std::ptrdiff_t get_available_stack_space()
std::size_t &get_continuation_recursion_count()
```

Protected Attributes

```
functor_type f_
context_state state_
thread_id_type id_
std::size_t thread_data_
std::size_t continuation_recursion_count_
```

Private Types

```
enum context_state
Values:
ctx_running
ctx_ready
ctx_exited
```

Private Functions

```
bool running() const
bool exited() const
```

Private Static Attributes

```
constexpr std::ptrdiff_t default_stack_size = -1
```

Friends

```
friend hpx::threads::coroutines::reset_on_exit
struct reset_on_exit
```

Public Functions

```
reset_on_exit(stackless_coroutine &this__)
~reset_on_exit()
```

Public Members

`stackless_coroutine &this_`

Defines

```
HPX_THREAD_STATE_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_PRIORITY_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_STATE_EX_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_STACKSIZE_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_SCHEDULE_HINT_UNSCOPED_ENUM_DEPRECATED_MSG
namespace hpx

namespace threads
```

Enums

`enum thread_schedule_state`

The `thread_schedule_state` enumerator encodes the current state of a `thread` instance

Values:

`unknown` = 0

`active` = 1

 thread is currently active (running, has resources)

`pending` = 2

 thread is pending (ready to run, but no hardware resource available)

`suspended` = 3

 thread has been suspended (waiting for synchronization event, but still known and under control
 of the thread-manager)

`depleted` = 4

 thread has been depleted (deeply suspended, it is not known to the thread-manager)

`terminated` = 5

 thread has been stopped 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:

`unknown` = -1

default_ = 0

Will assign the priority of the task to the default (normal) priority.

low = 1

Task goes onto a special low priority queue and will not be executed until all high/normal priority tasks are done, even if they are added after the low priority task.

normal = 2

Task will be executed when it is taken from the normal priority queue, this is usually a first in-first-out ordering of tasks (depending on scheduler choice). This is the default priority.

high_recursive = 3

The task is a high priority task and any child tasks spawned by this task will be made high priority as well - unless they are specifically flagged as non default priority.

boost = 4

Same as *thread_priority_high* except that the thread will fall back to *thread_priority_normal* if resumed after being suspended.

high = 5

Task goes onto a special high priority queue and will be executed before normal/low priority tasks are taken (some schedulers modify the behavior slightly and the documentation for those should be consulted).

bound = 6

Task goes onto a special high priority queue and will never be stolen by another thread after initial assignment. This should be used for thread placement tasks such as OpenMP type for loops.

enum thread_restart_state

The *thread_restart_state* enumerator encodes the reason why a thread is being restarted

Values:

unknown = 0**signaled = 1**

The thread has been signaled.

timeout = 2

The thread has been reactivated after a timeout

terminate = 3

The thread needs to be terminated.

abort = 4

The thread needs to be aborted.

enum thread_stacksize

A *thread_stacksize* references any of the possible stack-sizes for HPX threads.

Values:

unknown = -1**small_ = 1**

use small stack size (the underscore is to work around small being defined to char on Windows)

medium = 2

use medium sized stack size

large = 3

use large stack size

huge = 4
use very large stack size

nostack = 5
this thread does not suspend (does not need a stack)

current = 6
use size of current thread's stack

default_ = *small_*
use default stack size

minimal = *small_*
use minimally stack size

maximal = *huge*
use maximally stack size

enum thread_schedule_hint_mode
The type of hint given when creating new tasks.

Values:

none = 0
A hint that leaves the choice of scheduling entirely up to the scheduler.

thread = 1
A hint that tells the scheduler to prefer scheduling a task on the local thread number associated with this hint. Local thread numbers are indexed from zero. It is up to the scheduler to decide how to interpret thread numbers that are larger than the number of threads available to the scheduler. Typically thread numbers will wrap around when too large.

numa = 2
A hint that tells the scheduler to prefer scheduling a task on the NUMA domain associated with this hint. NUMA domains are indexed from zero. It is up to the scheduler to decide how to interpret NUMA domain indices that are larger than the number of available NUMA domains to the scheduler. Typically indices will wrap around when too large.

Functions

`std::ostream &operator<< (std::ostream &os, thread_schedule_state const t)`

`char const *get_thread_state_name (thread_schedule_state state)`

Returns the name of the given state.

Get the readable string representing the name of the given thread_state constant.

Parameters

- `state`: this represents the thread state.

`std::ostream &operator<< (std::ostream &os, thread_priority const t)`

`char const *get_thread_priority_name (thread_priority priority)`

Return the thread priority name.

Get the readable string representing the name of the given thread_priority constant.

Parameters

- `this`: represents the thread priority.

```
std::ostream &operator<< (std::ostream &os, thread_restart_state const t)

char const *get_thread_state_ex_name (thread_restart_state state)
    Get the readable string representing the name of the given thread_restart_state constant.

char const *get_thread_state_name (thread_state state)
    Get the readable string representing the name of the given thread_state constant.

std::ostream &operator<< (std::ostream &os, thread_stacksize const t)

char const *get_stack_size_enum_name (thread_stacksize size)
    Returns the stack size name.

    Get the readable string representing the given stack size constant.
```

Parameters

- `size`: this represents the stack size

```
hpx::threads::HPX_DEPRECATED_V(1, 6, HPX_THREAD_SCHEDULE_HINT_UNSCOPED_ENUM_DEPRECATED)
```

Variables

```
constexpr thread_schedule_state unknown = thread_schedule_state::unknown
constexpr thread_schedule_state active = thread_schedule_state::active
constexpr thread_schedule_state pending = thread_schedule_state::pending
constexpr thread_schedule_state suspended = thread_schedule_state::suspended
constexpr thread_schedule_state depleted = thread_schedule_state::depleted
constexpr thread_schedule_state terminated = thread_schedule_state::terminated
constexpr thread_schedule_state staged = thread_schedule_state::staged
constexpr thread_schedule_state pending_do_not_schedule = thread_schedule_state::pending_do_not_schedule
constexpr thread_schedule_state pending_boost = thread_schedule_state::pending_boost
constexpr thread_priority thread_priority_unknown = thread_priority::unknown
constexpr thread_priority thread_priority_default = thread_priority::default_
constexpr thread_priority thread_priority_low = thread_priority::low
constexpr thread_priority thread_priority_normal = thread_priority::normal
constexpr thread_priority thread_priority_high_recursive = thread_priority::high_recursive
constexpr thread_priority thread_priority_boost = thread_priority::boost
constexpr thread_priority thread_priority_high = thread_priority::high
constexpr thread_priority thread_priority_bound = thread_priority::bound
constexpr thread_priority thread_priority_critical = thread_priority::critical
constexpr thread_restart_state wait_unknown = thread_restart_state::unknown
constexpr thread_restart_state wait_signaled = thread_restart_state::signaled
constexpr thread_restart_state wait_timeout = thread_restart_state::timeout
constexpr thread_restart_state wait_terminate = thread_restart_state::terminate
```

```
constexpr thread_restart_state wait_abort = thread_restart_state::abort
constexpr thread_stacksize thread_stacksize_unknown = thread_stacksize::unknown
constexpr thread_stacksize thread_stacksize_small = thread_stacksize::small_
constexpr thread_stacksize thread_stacksize_medium = thread_stacksize::medium
constexpr thread_stacksize thread_stacksize_large = thread_stacksize::large
constexpr thread_stacksize thread_stacksize_huge = thread_stacksize::huge
constexpr thread_stacksize thread_stacksize_nostack = thread_stacksize::nostack
constexpr thread_stacksize thread_stacksize_current = thread_stacksize::current
constexpr thread_stacksize thread_stacksize_default = thread_stacksize::default_
constexpr thread_stacksize thread_stacksize_minimal = thread_stacksize::minimal
constexpr thread_stacksize thread_stacksize_maximal = thread_stacksize::maximal

struct thread_schedule_hint
    #include <thread_enums.hpp> A hint given to a scheduler to guide where a task should be scheduled.  
A scheduler is free to ignore the hint, or modify the hint to suit the resources available to the scheduler.
```

Public Functions

```
constexpr thread_schedule_hint()
    Construct a default hint with mode thread_schedule_hint_mode::none.  
  
constexpr thread_schedule_hint (std::int16_t thread_hint)
    Construct a hint with mode thread_schedule_hint_mode::thread and the given hint as the local  
thread number.  
  
constexpr thread_schedule_hint (thread_schedule_hint_mode mode, std::int16_t hint)
    Construct a hint with the given mode and hint. The numerical hint is unused when the mode is  
thread_schedule_hint_mode::none.
```

Public Members

```
std::int16_t hint
    The hint associated with the mode. The interpretation of this hint depends on the given mode.  
thread_schedule_hint_mode mode
    The mode of the scheduling hint.
```

```
template<>
struct hash<::hpx::threads::thread_id>
```

Public Functions

```
std::size_t operator() (::hpx::threads::thread_id const &v) const
template<>
struct hash<::hpx::threads::thread_id_ref>
```

Public Functions

```
std::size_t operator() (::hpx::threads::thread_id_ref const &v) const
namespace hpx
```

```
namespace threads
```

Enums

```
enum thread_id_addrf
```

Values:

`yes`

`no`

Variables

```
constexpr const thread_id invalid_thread_id
struct thread_id
```

Public Functions

```
thread_id()
```

```
thread_id(thread_id const&)
```

```
thread_id &operator=(thread_id const&)
```

```
constexpr thread_id(thread_id &&rhs)
```

```
constexpr thread_id &operator=(thread_id &&rhs)
```

```
constexpr thread_id(thread_id_repr const &thrd)
```

```
constexpr thread_id &operator=(thread_id_repr const &rhs)
```

```
constexpr operator bool() const
```

```
constexpr thread_id_repr get() const
```

```
constexpr void reset()
```

Private Types

```
using thread_id_repr = void*
```

Private Members

```
thread_id_repr thrd_ = nullptr
```

Friends

```
friend constexpr bool operator==(std::nullptr_t, thread_id const &rhs)
friend constexpr bool operator!=(std::nullptr_t, thread_id const &rhs)
friend constexpr bool operator==(thread_id const &lhs, std::nullptr_t)
friend constexpr bool operator!=(thread_id const &lhs, std::nullptr_t)
friend constexpr bool operator==(thread_id const &lhs, thread_id const &rhs)
friend constexpr bool operator!=(thread_id const &lhs, thread_id const &rhs)
friend constexpr bool operator<(thread_id const &lhs, thread_id const &rhs)
friend constexpr bool operator>(thread_id const &lhs, thread_id const &rhs)
friend constexpr bool operator<=(thread_id const &lhs, thread_id const &rhs)
friend constexpr bool operator>=(thread_id const &lhs, thread_id const &rhs)
template<typename Char, typename Traits>
std::basic_ostream<Char, Traits> &operator<< (std::basic_ostream<Char, Traits> &os,
                                                thread_id const &id)
void format_value(std::ostream &os, boost::string_ref spec, thread_id const &id)
```

```
struct thread_id_ref
```

Public Types

```
using thread_repr = detail::thread_data_reference_counting
```

Public Functions

```
thread_id_ref()
thread_id_ref(thread_id_ref const&)
thread_id_ref &operator=(thread_id_ref const&)
thread_id_ref (thread_id_ref &&rhs)
thread_id_ref &operator=(thread_id_ref &&rhs)
thread_id_ref (thread_id_repr const &thrd)
```

```

thread_id_ref(thread_id_repr &&thrd)
thread_id_ref &operator=(thread_id_repr const &rhs)
thread_id_ref &operator=(thread_id_repr &&rhs)
thread_id_ref(thread_repr *thrd, thread_id_addraddr = thread_id_addr::yes)
thread_id_ref &operator=(thread_repr *rhs)
thread_id_ref(thread_id const &noref)
thread_id_ref(thread_id &&noref)
thread_id_ref &operator=(thread_id const &noref)
thread_id_ref &operator=(thread_id &&noref)
operator bool() const
thread_id_noref() const
thread_id_repr &get() &
thread_id_repr &&get() &&
thread_id_repr const &get() const &
void reset()
void reset(thread_repr *thrd, bool add_ref = true)
constexpr thread_repr *detach()

```

Private Types

```
using thread_id_repr = hpx::intrusive_ptr<detail::thread_data_reference_counting>
```

Private Members

```
thread_id_repr thrd_
```

Friends

```

bool operator==(std::nullptr_t, thread_id_ref const &rhs)
bool operator!=(std::nullptr_t, thread_id_ref const &rhs)
bool operator==(thread_id_ref const &lhs, std::nullptr_t)
bool operator!=(thread_id_ref const &lhs, std::nullptr_t)
bool operator==(thread_id_ref const &lhs, thread_id_ref const &rhs)
bool operator!=(thread_id_ref const &lhs, thread_id_ref const &rhs)
bool operator<(thread_id_ref const &lhs, thread_id_ref const &rhs)

```

```
    bool operator> (thread_id_ref const &lhs, thread_id_ref const &rhs)
    bool operator<= (thread_id_ref const &lhs, thread_id_ref const &rhs)
    bool operator>= (thread_id_ref const &lhs, thread_id_ref const &rhs)
    template<typename Char, typename Traits>
        std::basic_ostream<Char, Traits> &operator<< (std::basic_ostream<Char, Traits> &os,
                                                       thread_id_ref const &id)
    void format_value (std::ostream &os, boost::string_ref spec, thread_id_ref const &id)

namespace std

template<>
struct hash<::hpx::threads::thread_id>
```

Public Functions

```
    std::size_t operator() (::hpx::threads::thread_id const &v) const
    template<>
    struct hash<::hpx::threads::thread_id_ref>
```

Public Functions

```
    std::size_t operator() (::hpx::threads::thread_id_ref const &v) const
```

datastructures

See *Public API* for a list of names and headers that are part of the public HPX API.

```
template<>
class basic_any<void, void, void, std::true_type>
```

Public Functions

```
    constexpr basic_any()
```

```
    basic_any(basic_any const &x)
```

```
    basic_any(basic_any &&x)
```

```
    template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>
        basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::type>::value >> type* = nullptr)
```

```
    template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts>
        basic_any(std::in_place_type_t<T>, Ts&&... ts)
```

```
    template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, U>
        basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)
```

```
    ~basic_any()
```

```

basic_any &operator=(basic_any const &x)

basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::ty
basic_any &operator=(T &&rhs)

basic_any &swap(basic_any &x)

std::type_info const &type () const

template<typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to(basic_any const &rhs) const

```

Private Functions

```
basic_any &assign(basic_any const &x)
```

Private Members

```

detail::any::fxn_ptr_table<void, void, void, std::true_type> *table
void *object

```

Private Static Functions

```

template<typename T, typename ...Ts>
static void new_object(void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static void new_object(void *&object, std::false_type, Ts&&... ts)

template<typename Char>
class basic_any<void, void, Char, std::true_type>

```

Public Functions

```

constexpr basic_any ()

basic_any (basic_any const &x)

basic_any (basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::ty
basic_any (T &&x, typename std::enable_if<std::is_copy_constructible<typename
std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts&&... ts>::value>::value>::type* = nullptr)
basic_any (std::in_place_type_t<T>, Ts&&... ts)

```

```
template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<type  
basic_any (std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)  
  
~basic_any ()  
  
basic_any &operator=(basic_any const &x)  
  
basic_any &operator=(basic_any &&rhs)  
  
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type  
basic_any &operator=(T &&rhs)  
  
basic_any &swap(basic_any &x)  
  
std::type_info const &type () const  
  
template<typename T>  
T const &cast () const  
  
bool has_value () const  
  
void reset ()  
  
bool equal_to (basic_any const &rhs) const
```

Private Functions

```
basic_any &assign(basic_any const &x)
```

Private Members

```
detail::any::fxn_ptr_table<void, void, Char, std::true_type> *table  
void *object
```

Private Static Functions

```
template<typename T, typename ...Ts>  
static void new_object (void *&object, std::true_type, Ts&&... ts)  
  
template<typename T, typename ...Ts>  
static void new_object (void *&object, std::false_type, Ts&&... ts)  
  
template<>  
class basic_any<void, void, void, std::false_type>
```

Public Functions

```

constexpr basic_any()

basic_any(basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value> basic_any(T &&x, typename std::enable_if<std::is_move_constructible<typename std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, std::in_place_type_t<T>, Ts&&... ts>::value> basic_any(std::in_place_type_t<T>, Ts&&... ts)

template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts>::value> basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

basic_any(basic_any const &x)

basic_any &operator=(basic_any const &x)

~basic_any()

basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value> basic_any &operator=(T &&rhs)

basic_any &swap(basic_any &x)

std::type_info const &type() const

template<typename T> T const &cast() const

bool has_value() const

void reset()

bool equal_to(basic_any const &rhs) const

```

Private Members

```

detail::any::fxn_ptr_table<void, void, void, std::false_type> *table
void *object

```

Private Static Functions

```

template<typename T, typename ...Ts> static void new_object(void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts> static void new_object(void *&object, std::false_type, Ts&&... ts)

template<typename Char> class basic_any<void, void, Char, std::false_type>

```

Public Functions

```
constexpr basic_any()

basic_any(basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value> T basic_any(T &&x, typename std::enable_if<std::is_move_constructible<typename std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts &&... ts>::value> T basic_any(std::in_place_type_t<T>, Ts&&... ts)

template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, std::initializer_list<U> il, Ts &&... ts>::value> T basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

basic_any(basic_any const &x)

basic_any &operator=(basic_any const &x)

~basic_any()

basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value> basic_any &operator=(T &&rhs)

basic_any &swap(basic_any &x)

std::type_info const &type() const

template<typename T>
T const &cast() const

bool has_value() const

void reset()

bool equal_to(basic_any const &rhs) const
```

Private Members

```
detail::any::fxn_ptr_table<void, void, Char, std::false_type> *table
void *object
```

Private Static Functions

```
template<typename T, typename ...Ts>
static void new_object(void *&object, std::true_type, Ts &&... ts)

template<typename T, typename ...Ts>
static void new_object(void *&object, std::false_type, Ts &&... ts)

namespace hpx
```

Typedefs

```
using any_nonser = util::basic_any<void, void, void, std::true_type>
using unique_any_nonser = util::basic_any<void, void, void, std::false_type>
```

Functions

```
template<typename T, typename ...Ts>
util::basic_any<void, void, void, std::true_type> make_any_nonser (Ts&&... ts)

template<typename T, typename U, typename ...Ts>
util::basic_any<void, void, void, std::true_type> make_any_nonser (std::initializer_list<U> il, Ts&&... ts)

template<typename T, typename ...Ts>
util::basic_any<void, void, void, std::false_type> make_unique_any_nonser (Ts&&... ts)

template<typename T, typename U, typename ...Ts>
util::basic_any<void, void, void, std::false_type> make_unique_any_nonser (std::initializer_list<U> il, Ts&&... ts)

template<typename T>
util::basic_any<void, void, void, std::true_type> make_any_nonser (T &&t)

template<typename T>
util::basic_any<void, void, void, std::false_type> make_unique_any_nonser (T &&t)

template<typename T, typename IArch, typename OArch, typename Char, typename Copyable>
T *any_cast (util::basic_any<IArch, OArch, Char, Copyable> *operand)

template<typename T, typename IArch, typename OArch, typename Char, typename Copyable>
T const *any_cast (util::basic_any<IArch, OArch, Char, Copyable> const *operand)

template<typename T, typename IArch, typename OArch, typename Char, typename Copyable>
T any_cast (util::basic_any<IArch, OArch, Char, Copyable> &operand)

template<typename T, typename IArch, typename OArch, typename Char, typename Copyable>
T const &any_cast (util::basic_any<IArch, OArch, Char, Copyable> const &operand)

struct bad_any_cast : public bad_cast
```

Public Functions

```
bad_any_cast (std::type_info const &src, std::type_info const &dest)
const char *what () const
```

Public Members

```
const char *from
const char *to

namespace util
```

TypeDefs

```
typedef hpx::tuple_element<I, T> instead

using streamable_any_nonser = basic_any<void, void, char, std::true_type>
using streamable_wany_nonser = basic_any<void, void, wchar_t, std::true_type>
using streamable_unique_any_nonser = basic_any<void, void, char, std::false_type>
using streamable_unique_wany_nonser = basic_any<void, void, wchar_t, std::false_type>
```

Functions

```
template<typename IArch, typename OArch, typename Char, typename Copyable, typename Enable = typename std::basic_istream<Char> &operator>> (std::basic_istream<Char> &i, basic_any<IArch, OArch, Char, Copyable> &obj)

template<typename IArch, typename OArch, typename Char, typename Copyable, typename Enable = typename std::basic_ostream<Char> &operator<< (std::basic_ostream<Char> &o, basic_any<IArch, OArch, Char, Copyable> const &obj)

template<typename IArch, typename OArch, typename Char, typename Copyable>
void swap(basic_any<IArch, OArch, Char, Copyable> &lhs, basic_any<IArch, OArch, Char, Copyable> &rhs)

template<typename T, typename... Ts> hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::")
std::true_type make_any_nonser(Ts&&... ts)

template<typename T, typename U, typename... Ts> hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::")
std::true_type make_any_nonser(std::initializer_list<U> il, Ts&&... ts)

template<typename T, typename Char, typename... Ts>
basic_any<void, void, Char, std::true_type> make_streamable_any_nonser(Ts&&... ts)

template<typename T, typename Char, typename U, typename... Ts>
basic_any<void, void, Char, std::true_type> make_streamable_any_nonser(std::initializer_list<U> il, Ts&&... ts)

template<typename T, typename... Ts> hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::")
std::false_type make_unique_any_nonser(Ts&&... ts)

template<typename T, typename U, typename... Ts> hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::")
std::false_type make_unique_any_nonser(std::initializer_list<U> il, Ts&&... ts)

template<typename T, typename Char, typename... Ts>
basic_any<void, void, Char, std::false_type> make_streamable_unique_any_nonser(Ts&&... ts)
```

```

template<typename T, typename Char, typename U, typename ...Ts>
basic_any<void, void, Char, std::false_type> make_streamable_unique_any_nonser (std::initializer_list<U>
il,
Ts&&...
ts)

template<typename T>hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_any_nonser i

std::true_type make_any_nonser (T &&t)
template<typename T, typename Char>
basic_any<void, void, Char, std::true_type> make_streamable_any_nonser (T &&t)
template<typename T>hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_unique_any_n

std::false_type make_unique_any_nonser (T &&t)
template<typename T, typename Char>
basic_any<void, void, Char, std::false_type> make_streamable_unique_any_nonser (T
&&t)

```

Variables

```

hpx::util::void

template<typename Char>
class basic_any<void, void, Char, std::false_type>

```

Public Functions

```

constexpr basic_any()

basic_any(basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>>::value>>::type* = nullptr)
basic_any(T &&x, typename std::enable_if<std::is_move_constructible<typename std::decay<T>::type>>::value>>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename basic_any, std::in_place_type_t<T>, Ts&&... ts>>::value>>::type* = nullptr

template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename basic_any, std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts>>::value>>::type* = nullptr

basic_any(basic_any const &x)

basic_any &operator=(basic_any const &x)

~basic_any()

basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>>::value>>::type* = nullptr)
basic_any &operator=(T &&rhs)

basic_any &swap(basic_any &x)

std::type_info const &type() const

```

```
template<typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const &rhs) const
```

Private Members

```
detail::any::fxn_ptr_table<void, void, Char, std::false_type> *table
void *object
```

Private Static Functions

```
template<typename T, typename ...Ts>
static void new_object (void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static void new_object (void *&object, std::false_type, Ts&&... ts)

template<typename Char>
class basic_any<void, void, Char, std::true_type>
```

Public Functions

```
constexpr basic_any ()

basic_any (basic_any const &x)

basic_any (basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value*>::type* = nullptr)
basic_any (T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::value>::value*>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::value, Ts&&... ts>::value*>::type* = nullptr)
basic_any (std::in_place_type_t<T>, Ts&&... ts)

template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<basic_any, std::initializer_list<U> il, Ts&&... ts>::value*>::type* = nullptr)
basic_any (std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

~basic_any ()

basic_any &operator= (basic_any const &x)

basic_any &operator= (basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value*>::type* = nullptr)
basic_any &operator= (T &&rhs)

basic_any &swap (basic_any &x)

std::type_info const &type () const
```

```
template<typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const &rhs) const
```

Private Functions

```
basic_any &assign (basic_any const &x)
```

Private Members

```
detail::any::fxn_ptr_table<void, void, Char, std::true_type> *table
void *object
```

Private Static Functions

```
template<typename T, typename ...Ts>
static void new_object (void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static void new_object (void *&object, std::false_type, Ts&&... ts)

template<>
class basic_any<void, void, void, std::false_type>
```

Public Functions

```
constexpr basic_any()

basic_any (basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value* = nullptr>
basic_any (T &&x, typename std::enable_if<std::is_move_constructible<typename std::decay<T>::value>::value* = nullptr>
```

```
template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::value, Ts>::value>::value* = nullptr>
basic_any (std::in_place_type_t<T>, Ts&&... ts)
```

```
template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<basic_any, U>::value>::value* = nullptr>
basic_any (std::in_place_type_t<U>, std::initializer_list<U> il, Ts&&... ts)
```

```
basic_any (basic_any const &x)

basic_any &operator= (basic_any const &x)

~basic_any()

basic_any &operator= (basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value* = nullptr>
basic_any &operator= (basic_any &&rhs)
```

```
basic_any &operator= (T &&rhs)
basic_any &swap (basic_any &x)
std::type_info const &type () const
template<typename T>
T const &cast () const
bool has_value () const
void reset ()
bool equal_to (basic_any const &rhs) const
```

Private Members

```
detail::any::fxn_ptr_table<void, void, void, std::false_type> *table
void *object
```

Private Static Functions

```
template<typename T, typename ...Ts>
static void new_object (void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static void new_object (void *&object, std::false_type, Ts&&... ts)

template<>
class basic_any<void, void, void, std::true_type>
```

Public Functions

```
constexpr basic_any ()

basic_any (basic_any const &x)

basic_any (basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value>::type* = nullptr>
basic_any (T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::value>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename basic_any (std::in_place_type_t<T>, Ts&&... ts)

template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename basic_any (std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

~basic_any ()

basic_any &operator= (basic_any const &x)

basic_any &operator= (basic_any &&rhs)
```

```

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<basic_any>&operator=(T &&rhs)

basic_any &swap (basic_any &x)

std::type_info const &type () const

template<typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const &rhs) const

```

Private Functions

```
basic_any &assign (basic_any const &x)
```

Private Members

```
detail::any::fxn_ptr_table<void, void, void, std::true_type> *table
void *object
```

Private Static Functions

```

template<typename T, typename ...Ts>
static void new_object (void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static void new_object (void *&object, std::false_type, Ts&&... ts)

template<std::size_t... Is, typename ...Ts>
struct member_pack<util::index_pack<Is...>, Ts...> : public hpx::util::detail::member_leaf<Is, Ts>

```

Public Functions

```

member_pack ()

template<typename ...Us>
constexpr member_pack (std::piecewise_construct_t, Us&&... us)

template<std::size_t I>
constexpr decltype(auto) get () &

template<std::size_t I>
constexpr decltype(auto) get () const &

template<std::size_t I>
constexpr decltype(auto) get () &&

template<std::size_t I>
constexpr decltype(auto) get () const &&

```

```
namespace hpx

namespace serialization
```

Functions

```
template<typename Archive, std::size_t... Is, typename ...Ts>
void serialize(Archive &ar, ::hpx::util::member_pack<util::index_pack<Is..., Ts...>, &mp, un-
signed int const = 0)
```

```
namespace util
```

Typedefs

```
template<typename ...Ts>
using member_pack_for = member_pack<typename util::make_index_pack<sizeof...(Ts)>::type, Ts...>
```

Variables

```
template<typename Is, typename ...Ts>
struct HPX_EMPTY_BASES member_pack

template<std::size_t... Is, typename ...Ts>
struct member_pack<util::index_pack<Is..., Ts...>, Ts...> : public hpx::util::detail::member_leaf<Is, Ts>
```

Public Functions

```
member_pack()

template<typename ...Us>
constexpr member_pack(std::piecewise_construct_t, Us&&... us)

template<std::size_t I>
constexpr decltype(auto) get() &

template<std::size_t I>
constexpr decltype(auto) get() const &

template<std::size_t I>
constexpr decltype(auto) get() const &&

template<std::size_t I>
constexpr decltype(auto) get() const &&

template<typename T>
struct hash<hpx::optional<T>>
```

Public Functions

```
constexpr std::size_t operator() (::hpx::optional<T> const &arg) const
namespace hpx

namespace util
```

Functions

```
template<typename T>
constexpr bool operator==(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator!=(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator<(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator>=(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator>(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator<=(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator==(optional<T> const &opt, nullopt_t)

template<typename T>
constexpr bool operator==(nullopt_t, optional<T> const &opt)

template<typename T>
constexpr bool operator!=(optional<T> const &opt, nullopt_t)

template<typename T>
constexpr bool operator!=(nullopt_t, optional<T> const &opt)

template<typename T>
constexpr bool operator<(optional<T> const&, nullopt_t)

template<typename T>
constexpr bool operator<(nullopt_t, optional<T> const &opt)

template<typename T>
constexpr bool operator>=(optional<T> const&, nullopt_t)

template<typename T>
constexpr bool operator>=(nullopt_t, optional<T> const &opt)

template<typename T>
constexpr bool operator>(optional<T> const &opt, nullopt_t)

template<typename T>
```

```
constexpr bool operator> (nullopt_t, optional<T> const&)

template<typename T>
constexpr bool operator<= (optional<T> const &opt, nullopt_t)

template<typename T>
constexpr bool operator<= (nullopt_t, optional<T> const&)

template<typename T>
constexpr bool operator== (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator== (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator!= (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator!= (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator< (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator< (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator>= (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator>= (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator> (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator> (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator<= (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator<= (T const &value, optional<T> const &opt)

template<typename T>
void swap (optional<T> &x, optional<T> &y)

template<typename T>
constexpr optional<typename std::decay<T>::type> make_optional (T &&v)

template<typename T, typename ...Ts>
constexpr optional<T> make_optional (Ts&&... ts)

template<typename T, typename U, typename ...Ts>
constexpr optional<T> make_optional (std::initializer_list<U> il, Ts&&... ts)
```

Variables

```
constexpr nullopt_t nullopt = {nullopt_t::init()}

class bad_optional_access : public logic_error
```

Public Functions

```
bad_optional_access (std::string const &what_arg)

bad_optional_access (char const *what_arg)

struct nullopt_t
```

Public Functions

```
constexpr nullopt_t (nullopt_t::init)

template<typename T>
class optional
```

Public Types

```
template<>
using value_type = T
```

Public Functions

```
constexpr optional()

constexpr optional (nullopt_t)

optional (optional const &other)

optional (optional &&other)

optional (T const &val)

optional (T &&val)

template<typename ...Ts>
optional (in_place_t, Ts&&... ts)

template<typename U, typename ...Ts>
optional (in_place_t, std::initializer_list<U> il, Ts&&... ts)

~optional()

optional &operator= (optional const &other)

optional &operator= (optional &&other)

optional &operator= (T const &other)

optional &operator= (T &&other)
```

```
optional &operator=(nullopt_t)

constexpr T const *operator->() const

T *operator->()

constexpr T const &operator*() const

T &operator*()

constexpr operator bool() const

constexpr bool has_value() const

T &value()

T const &value() const

template<typename U>
constexpr T value_or(U &&value) const

template<typename ...Ts>
void emplace(Ts&&... ts)

template<typename F, typename ...Ts>
void emplace_f(F &&f, Ts&&... ts)

void swap(optional &other)

void reset()
```

Private Members

```
std::aligned_storage<sizeof(T), alignof(T)>::type storage_
bool empty_
namespace _optional_swap
```

Functions

```
template<typename T>
void check_swap()

namespace std

template<typename T>
struct hash<hpx::optional<T>>
```

Public Functions

```
constexpr std::size_t operator() (::hpx::optional<T> const &arg) const
template<typename T0, typename T1>
struct tuple_element<0, std::pair<T0, T1>>
```

Public Types

```
template<>
using type = T0
```

Public Static Functions

```
static constexpr type &get (std::pair<T0, T1> &tuple)
static constexpr type const &get (std::pair<T0, T1> const &tuple)
```

```
template<typename T0, typename T1>
struct tuple_element<1, std::pair<T0, T1>>
```

Public Types

```
template<>
using type = T1
```

Public Static Functions

```
static constexpr type &get (std::pair<T0, T1> &tuple)
static constexpr type const &get (std::pair<T0, T1> const &tuple)
```

```
template<std::size_t I, typename Type, std::size_t Size>
struct tuple_element<I, std::array<Type, Size>>
```

Public Types

```
template<>
using type = Type
```

Public Static Functions

```
static constexpr type &get (std::array<Type, Size> &tuple)
static constexpr type const &get (std::array<Type, Size> const &tuple)
```

```
namespace hpx
```

Functions

```
template<typename ...Ts>
constexpr tuple<typename util::decay_unwrap<Ts>::type...> make_tuple (Ts&&... vs)

template<typename ...Ts>
tuple<Ts&&...> forward_as_tuple (Ts&&... vs)

template<typename ...Ts>
tuple<Ts&...> tie (Ts&... vs)

template<typename ...Tuples>
constexpr auto tuple_cat (Tuples&&... tuples)

HPX_FORWARD (Tuples, tuples)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator== (tuple<Ts...>
const &t,
tuple<Us...>
const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator!= (tuple<Ts...>
const &t,
tuple<Us...>
const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator< (tuple<Ts...>
const &t, tu-
ple<Us...> const
&u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator> (tuple<Ts...>
const &t, tu-
ple<Us...> const
&u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator<= (tuple<Ts...>
const &t,
tuple<Us...>
const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator>= (tuple<Ts...>
const &t,
tuple<Us...>
const &u)

template<typename ...Ts>
void swap (tuple<Ts...> &x, tuple<Ts...> &y)
```

Variables

```
const hpx::detail::ignore_type ignore = {}  
  
template<typename ...Ts>  
class tuple
```

Public Functions

```
template<typename Dependent = void, typename Enable = typename std::enable_if<util::all_of<std::is_constructible<  
constexpr tuple ()  
  
constexpr tuple (Ts const&... vs)  
  
template<typename U, typename ...Us, typename Enable = typename std::enable_if<!std::is_same<tuple, typename std::decay<  
constexpr tuple (U &&v, Us&&... vs)  
  
tuple (tuple const&)  
  
tuple (tuple&&)  
  
template<typename UTuple, typename Enable = typename std::enable_if<!std::is_same<tuple, typename std::decay<  
constexpr tuple (UTuple &&other)  
  
tuple &operator= (tuple const &other)  
  
tuple &operator= (tuple &&other)  
  
template<typename UTuple>  
tuple &operator= (UTuple &&other)  
  
void swap (tuple &other)  
  
template<std::size_t I>  
util::at_index<I, Ts...>::type &get ()  
  
template<std::size_t I>  
util::at_index<I, Ts...>::type const &get () const
```

Private Types

```
template<>  
using index_pack = typename util::make_index_pack<sizeof...(Ts)>::type
```

Private Functions

```
template<std::size_t... Is, typename UTuple>  
constexpr tuple (util::index_pack<Is...>, UTuple &&other)  
  
template<std::size_t... Is>  
void assign_ (util::index_pack<Is...>, tuple const &other)  
  
template<std::size_t... Is>  
void assign_ (util::index_pack<Is...>, tuple &&other)
```

```
template<std::size_t... Is, typename UTuple>
void assign_(util::index_pack<Is...>, UTuple &&other)

template<std::size_t... Is>
void swap_(util::index_pack<Is...>, tuple &&other)
```

Private Members

```
util::member_pack_for<Ts...> _members
```

```
template<>
class tuple<>
```

Public Functions

```
constexpr tuple()
constexpr tuple(tuple const&)
constexpr tuple(tuple&&)
tuple &operator=(tuple const&)
tuple &operator=(tuple&&)
void swap(tuple&)
```

```
template<typename T0, typename T1>
struct tuple_element<0, std::pair<T0, T1>>
```

Public Types

```
template<>
using type = T0
```

Public Static Functions

```
static constexpr type &get(std::pair<T0, T1> &tuple)
static constexpr type const &get(std::pair<T0, T1> const &tuple)

template<typename T0, typename T1>
struct tuple_element<1, std::pair<T0, T1>>
```

Public Types

```
template<>
using type = T1
```

Public Static Functions

```
static constexpr type &get (std::pair<T0, T1> &tuple)
static constexpr type const &get (std::pair<T0, T1> const &tuple)
```

```
template<std::size_t I, typename Type, std::size_t Size>
struct tuple_element<I, std::array<Type, Size>>
```

Public Types

```
template<>
using type = Type
```

Public Static Functions

```
static constexpr type &get (std::array<Type, Size> &tuple)
static constexpr type const &get (std::array<Type, Size> const &tuple)
```

```
template<std::size_t I, typename ...Ts>
struct tuple_element<I, tuple<Ts...>>
```

Public Types

```
template<>
using type = typename util::at_index::type
```

Public Static Functions

```
static constexpr type &get (hpx::tuple<Ts...> &tuple)
static constexpr type const &get (hpx::tuple<Ts...> const &tuple)
```

```
template<class T>
struct tuple_size
    Subclassed by hpx::tuple_size< const T >, hpx::tuple_size< const volatile T >, hpx::tuple_size< volatile T >
```

```
namespace adl_barrier
```

Functions

```
template<std::size_t I, typename Tuple, typename Enable = typename util::always_void<typename hpx::tuple_element<I, Tuple>::type &> get(Tuple &t)  
  
template<std::size_t I, typename Tuple, typename Enable = typename util::always_void<typename hpx::tuple_element<I, Tuple>::type const &> get(Tuple const &t)  
  
template<std::size_t I, typename Tuple, typename Enable = typename util::always_void<typename hpx::tuple_element<I, Tuple>::type && > get(Tuple &&t)  
  
template<std::size_t I, typename Tuple, typename Enable = typename util::always_void<typename hpx::tuple_element<I, Tuple>::type const && > get(Tuple const &&t)  
  
template<std::size_t I, typename Tuple, typename Enable>  
constexpr tuple_element<I, Tuple>::type & get(Tuple &t)  
  
template<std::size_t I, typename Tuple, typename Enable>  
constexpr tuple_element<I, Tuple>::type const & get(Tuple const &t)  
  
template<std::size_t I, typename Tuple, typename Enable>  
constexpr tuple_element<I, Tuple>::type && get(Tuple &&t)  
  
template<std::size_t I, typename Tuple, typename Enable>  
constexpr tuple_element<I, Tuple>::type const && get(Tuple const &&t)  
  
namespace std_adl_barrier
```

Functions

```
template<std::size_t I, typename ...Ts>  
constexpr hpx::tuple_element<I, hpx::tuple<Ts...>::type &> get(hpx::tuple<Ts...> &t)  
  
template<std::size_t I, typename ...Ts>  
constexpr hpx::tuple_element<I, hpx::tuple<Ts...>::type const &> get(hpx::tuple<Ts...>  
const &t)  
  
template<std::size_t I, typename ...Ts>  
constexpr hpx::tuple_element<I, hpx::tuple<Ts...>::type && > get(hpx::tuple<Ts...> &&t)  
  
template<std::size_t I, typename ...Ts>  
constexpr hpx::tuple_element<I, hpx::tuple<Ts...>::type const && > get(hpx::tuple<Ts...>  
const &&t)  
  
template<std::size_t I, typename ...Ts>  
constexpr tuple_element<I, tuple<Ts...>::type &> get(tuple<Ts...> &t)  
  
template<std::size_t I, typename ...Ts>  
constexpr tuple_element<I, tuple<Ts...>::type const &> get(tuple<Ts...> const &t)  
  
template<std::size_t I, typename ...Ts>  
constexpr tuple_element<I, tuple<Ts...>::type && > get(tuple<Ts...> &&t)  
  
template<std::size_t I, typename ...Ts>  
constexpr tuple_element<I, tuple<Ts...>::type const && > get(tuple<Ts...> const &&t)  
  
namespace util
```

Functions

```
hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::ignore is deprecated. Use hpx::ignore")
template<typename... Ts>hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_tuple is")
template<typename... Ts>hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::tie is depre")
template<typename... Ts>hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::forward_as_tu")
template<typename... Ts>hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::tuple_cat is")
template<std::size_t I, typename Tuple>hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util")
namespace hpx
```

```
namespace adl_barrier
```

Functions

```
template<std::size_t I, typename ...Ts>
constexpr util::at_index<I, Ts...>::type &get (std::variant<Ts...> &var)

template<std::size_t I, typename ...Ts>
constexpr util::at_index<I, Ts...>::type const &get (std::variant<Ts...> const &var)

template<std::size_t I, typename ...Ts>
constexpr util::at_index<I, Ts...>::type &&get (std::variant<Ts...> &&var)

template<std::size_t I, typename ...Ts>
constexpr util::at_index<I, Ts...>::type const &&get (std::variant<Ts...> const &&var)
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename T>
struct is_tuple_like : public hpx::traits::detail::is_tuple_like_impl<std::remove_cv<T>::type>
    #include <is_tuple_like.hpp> Deduces to a true type if the given parameter T has a specific tuple like
    size.
```

debugging

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

Functions

```
void attach_debugger()
```

Tries to break an attached debugger, if not supported a loop is invoked which gives enough time to attach a debugger manually.

```
namespace hpx
```

```
namespace util
```

Functions

```
std::string trace (std::size_t frames_no = HPX_HAVE_THREAD_BACKTRACE_DEPTH)
```

```
namespace hpx
```

```
namespace util
```

```
namespace debug
```

Typedefs

```
template<typename T>
using cxxabi_demangle_helper = demangle_helper<T>

template<typename T>
using cxx_type_id = type_id<T>
```

Functions

```
template<typename T = void>
std::string print_type (const char* = "")
```

```
template<>
std::string print_type (const char *delim)
```

```
template<typename T>
struct demangle_helper
```

Public Functions

```
char const *type_id () const
```

```
template<typename T>
struct type_id
```

Public Static Attributes

```
demangle_helper<T> typeid_ = demangle_helper<T>()
```

Variables

```
char **environ
```

Defines

```
HPX_DP_LAZY (Expr, printer)
```

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename E>  
details::trace_manip trace (E const &e)
```

```
namespace details
```

Functions

```
std::ostream &operator<< (std::ostream &out, details::trace_manip const &t)
```

```
class trace_manip
```

Public Functions

```
trace_manip (backtrace const *tr)
```

```
std::ostream &write (std::ostream &out) const
```

Private Members

```
backtrace const *tr_
```

```
namespace stack_trace
```

Functions

```
std::size_t trace (void **addresses, std::size_t size)  
void write_symbols (void *const *addresses, std::size_t size, std::ostream&)  
std::string get_symbol (void *address)  
std::string get_symbols (void *const *address, std::size_t size)
```

errors

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

Enums

enum error

Possible error conditions.

This enumeration lists all possible error conditions which can be reported from any of the API functions.

Values:

success = 0

The operation was successful.

no_success = 1

The operation did failed, but not in an unexpected manner.

not_implemented = 2

The operation is not implemented.

out_of_memory = 3

The operation caused an out of memory condition.

bad_action_code = 4

bad_component_type = 5

The specified component type is not known or otherwise invalid.

network_error = 6

A generic network error occurred.

version_too_new = 7

The version of the network representation for this object is too new.

version_too_old = 8

The version of the network representation for this object is too old.

version_unknown = 9

The version of the network representation for this object is unknown.

unknown_component_address = 10

duplicate_component_address = 11

The given global id has already been registered.

invalid_status = 12

The operation was executed in an invalid status.

```
bad_parameter = 13
    One of the supplied parameters is invalid.

internal_server_error = 14

service_unavailable = 15

bad_request = 16

repeated_request = 17

lock_error = 18

duplicate_console = 19
    There is more than one console locality.

no_registered_console = 20
    There is no registered console locality available.

startup_timed_out = 21

uninitialized_value = 22

bad_response_type = 23

deadlock = 24

assertion_failure = 25

null_thread_id = 26
    Attempt to invoke a API function from a non-HPX thread.

invalid_data = 27

yield_aborted = 28
    The yield operation was aborted.

dynamic_link_failure = 29

commandline_option_error = 30
    One of the options given on the command line is erroneous.

serialization_error = 31
    There was an error during serialization of this object.

unhandled_exception = 32
    An unhandled exception has been caught.

kernel_error = 33
    The OS kernel reported an error.

broken_task = 34
    The task associated with this future object is not available anymore.

task_moved = 35
    The task associated with this future object has been moved.

task_already_started = 36
    The task associated with this future object has already been started.

future_already_retrieved = 37
    The future object has already been retrieved.

promise_already_satisfied = 38
    The value for this future object has already been set.
```

```
future_does_not_support_cancellation = 39
    The future object does not support cancellation.

future_can_not_be_cancelled = 40
    The future can't be canceled at this time.

no_state = 41
    The future object has no valid shared state.

broken.promise = 42
    The promise has been deleted.

thread_resource_error = 43

future_cancelled = 44

thread_cancelled = 45

thread_not_interruptable = 46

duplicate_component_id = 47
    The component type has already been registered.

unknown_error = 48
    An unknown error occurred.

bad_plugin_type = 49
    The specified plugin type is not known or otherwise invalid.

filesystem_error = 50
    The specified file does not exist or other filesystem related error.

bad_function_call = 51
    equivalent of std::bad_function_call

task_canceled_exception = 52
    parallel::v2::task_canceled_exception

task_block_not_active = 53
    task_region is not active

out_of_range = 54
    Equivalent to std::out_of_range.

length_error = 55
    Equivalent to std::length_error.

migration_needs_retry = 56
    migration failed because of global race, retry

namespace hpx
```

Unnamed Group

```
error_code make_error_code(error e, throwmode mode = plain)
    Returns a new error_code constructed from the given parameters.
```

```
error_code make_error_code(error e, char const *func, char const *file, long line, throwmode
    mode = plain)
```

```
error_code make_error_code(error e, char const *msg, throwmode mode = plain)
    Returns error_code(e, msg, mode).
```

```
error_code make_error_code(error e, char const *msg, char const *func, char const *file, long
                           line, throwmode mode = plain)

error_code make_error_code(error e, std::string const &msg, throwmode mode = plain)
                           Returns error_code(e, msg, mode).

error_code make_error_code(error e, std::string const &msg, char const *func, char const
                           *file, long line, throwmode mode = plain)

error_code make_error_code(std::exception_ptr const &e)
```

Functions

`std::error_category const &get_hpx_category()`
 Returns generic HPX error category used for new errors.

`std::error_category const &get_hpx_rethrow_category()`
 Returns generic HPX error category used for errors re-thrown after the exception has been de-serialized.

`error_code make_success_code(throwmode mode = plain)`
 Returns `error_code(hpx::success, "success", mode)`.

`class error_code : public error_code`
`#include <error_code.hpp>` A `hpx::error_code` represents an arbitrary error condition.

The class `hpx::error_code` describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces.

Note Class `hpx::error_code` is an adjunct to error reporting by exception

Public Functions

`error_code(throwmode mode = plain)`
 Construct an object of type `error_code`.

Parameters

- mode: The parameter mode specifies whether the constructed `hpx::error_code` belongs to the error category `hpx_category` (if mode is `plain`, this is the default) or to the category `hpx_category_rethrow` (if mode is `rethrow`).

Exceptions

- nothing:

`error_code(error e, throwmode mode = plain)`
 Construct an object of type `error_code`.

Parameters

- e: The parameter e holds the `hpx::error_code` the new exception should encapsulate.
- mode: The parameter mode specifies whether the constructed `hpx::error_code` belongs to the error category `hpx_category` (if mode is `plain`, this is the default) or to the category `hpx_category_rethrow` (if mode is `rethrow`).

Exceptions

- nothing:

error_code (*error e, char const *func, char const *file, long line, throwmode mode = plain*)
Construct an object of type *error_code*.

Parameters

- *e*: The parameter *e* holds the hpx::error code the new exception should encapsulate.
- *func*: The name of the function where the error was raised.
- *file*: The file name of the code where the error was raised.
- *line*: The line number of the code line where the error was raised.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- *nothing*:

error_code (*error e, char const *msg, throwmode mode = plain*)
Construct an object of type *error_code*.

Parameters

- *e*: The parameter *e* holds the hpx::error code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- *std::bad_alloc*: (if allocation of a copy of the passed string fails).

error_code (*error e, char const *msg, char const *func, char const *file, long line, throwmode mode = plain*)
Construct an object of type *error_code*.

Parameters

- *e*: The parameter *e* holds the hpx::error code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *func*: The name of the function where the error was raised.
- *file*: The file name of the code where the error was raised.
- *line*: The line number of the code line where the error was raised.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- *std::bad_alloc*: (if allocation of a copy of the passed string fails).

error_code (*error e, std::string const &msg, throwmode mode = plain*)
Construct an object of type *error_code*.

Parameters

- *e*: The parameter *e* holds the hpx::error code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- *std::bad_alloc*: (if allocation of a copy of the passed string fails).

```
error_code (error e, std::string const &msg, char const *func, char const *file, long line,  
    throwmode mode = plain)  
Construct an object of type error_code.
```

Parameters

- *e*: The parameter *e* holds the hpx::error code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *func*: The name of the function where the error was raised.
- *file*: The file name of the code where the error was raised.
- *line*: The line number of the code line where the error was raised.
- *mode*: The parameter *mode* specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if *mode* is *plain*, this is the default) or to the category *hpx_category_rethrow* (if *mode* is *rethrow*).

Exceptions

- *std::bad_alloc*: (if allocation of a copy of the passed string fails).

```
std::string get_message () const
```

Return a reference to the error message stored in the *hpx::error_code*.

Exceptions

- *nothing*:

```
void clear ()
```

Clear this *error_code* object. The postconditions of invoking this method are.

- *value() == hpx::success* and *category() == hpx::get_hpx_category()*

```
error_code (error_code const &rhs)
```

Copy constructor for *error_code*

Note This function maintains the error category of the left hand side if the right hand side is a success code.

```
error_code &operator= (error_code const &rhs)
```

Assignment operator for *error_code*

Note This function maintains the error category of the left hand side if the right hand side is a success code.

Private Functions

```
error_code (int err, hpx::exception const &e)
```

```
error_code (std::exception_ptr const &e)
```

Private Members

```
std::exception_ptr exception_
```

Friends

```
friend hpx::exception  
error_code make_error_code (std::exception_ptr const &e)  
namespace hpx
```

Typedefs

```
using custom_exception_info_handler_type = std::function<hpx::exception_info (std::string  
const&,  
std::string  
const&,  
long,  
std::string  
const&) >  
using pre_exception_handler_type = std::function<void () >
```

Functions

```
void set_custom_exception_info_handler (custom_exception_info_handler_type f)
```

```
void set_pre_exception_handler (pre_exception_handler_type f)
```

```
std::string get_error_what (exception_info const &xi)
```

Return the error message of the thrown exception.

The function `hpx::get_error_what` can be used to extract the diagnostic information element representing the error message as stored in the given exception instance.

Return The error message stored in the exception If the exception instance does not hold this information, the function will return an empty string.

See `hpx::diagnostic_information()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`, `hpx::get_error_function_name()`, `hpx::get_error_file_name()`, `hpx::get_error_line_number()`, `hpx::get_error_os_thread()`, `hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`, `hpx::get_error()` `hpx::get_error_backtrace()`, `hpx::get_error_env()`, `hpx::get_error_config()`, `hpx::get_error_state()`

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception_info`, `hpx::error_code`, `std::exception`, or `std::exception_ptr`.

Exceptions

- `std::bad_alloc`: (if one of the required allocations fails)

`error get_error (hpx::exception const &e)`

Return the error code value of the exception thrown.

The function `hpx::get_error` can be used to extract the diagnostic information element representing the error value code as stored in the given exception instance.

Return The error value code of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return `hpx::naming::invalid_locality_id`.

See `hpx::diagnostic_information()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`,
`hpx::get_error_function_name()`, `hpx::get_error_file_name()`, `hpx::get_error_line_number()`,
`hpx::get_error_os_thread()`, `hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`,
`hpx::get_error_backtrace()`, `hpx::get_error_env()`, `hpx::get_error_what()`, `hpx::get_error_config()`,
`hpx::get_error_state()`

Parameters

- `e`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception`, `hpx::error_code`, or `std::exception_ptr`.

Exceptions

- nothing:

`error get_error (hpx::error_code const &e)`

`std::string get_error_function_name (hpx::exception_info const &xi)`

Return the function name from which the exception was thrown.

The function `hpx::get_error_function_name` can be used to extract the diagnostic information element representing the name of the function as stored in the given exception instance.

Return The name of the function from which the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

See `hpx::diagnostic_information()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`,
`hpx::get_error_file_name()`, `hpx::get_error_line_number()`, `hpx::get_error_os_thread()`,
`hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`, `hpx::get_error()`,
`hpx::get_error_backtrace()`, `hpx::get_error_env()`, `hpx::get_error_what()`, `hpx::get_error_config()`,
`hpx::get_error_state()`

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception_info`, `hpx::error_code`, `std::exception`, or `std::exception_ptr`.

Exceptions

- `std::bad_alloc`: (if one of the required allocations fails)

`std::string get_error_file_name (hpx::exception_info const &xi)`

Return the (source code) file name of the function from which the exception was thrown.

The function `hpx::get_error_file_name` can be used to extract the diagnostic information element representing the name of the source file as stored in the given exception instance.

Return The name of the source file of the function from which the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

See `hpx::diagnostic_information()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`,
`hpx::get_error_function_name()`, `hpx::get_error_line_number()`, `hpx::get_error_os_thread()`,
`hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`, `hpx::get_error()`,
`hpx::get_error_backtrace()`, `hpx::get_error_env()`, `hpx::get_error_what()`, `hpx::get_error_config()`,
`hpx::get_error_state()`

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception_info`, `hpx::error_code`, `std::exception`, or `std::exception_ptr`.

Exceptions

- `std::bad_alloc`: (if one of the required allocations fails)

long **get_error_line_number** (`hpx::exception_info const &xi`)

Return the line number in the (source code) file of the function from which the exception was thrown.

The function `hpx::get_error_line_number` can be used to extract the diagnostic information element representing the line number as stored in the given exception instance.

Return The line number of the place where the exception was thrown. If the exception instance does not hold this information, the function will return -1.

See `hpx::diagnostic_information()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`,
`hpx::get_error_function_name()`, `hpx::get_error_file_name()`, `hpx::get_error_os_thread()`,
`hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`, `hpx::get_error()`,
`hpx::get_error_backtrace()`, `hpx::get_error_env()`, `hpx::get_error_what()`, `hpx::get_error_config()`,
`hpx::get_error_state()`

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception_info`, `hpx::error_code`, `std::exception`, or `std::exception_ptr`.

Exceptions

- nothing:

class exception : public system_error

#include <exception.hpp> A `hpx::exception` is the main exception type used by HPX to report errors.

The `hpx::exception` type is the main exception type used by HPX to report errors. Any exceptions thrown by functions in the HPX library are either of this type or of a type derived from it. This implies that it is always safe to use this type only in catch statements guarding HPX library calls.

Subclassed by `hpx::exception_list`

Public Functions

exception (*error e = success*)

Construct a *hpx::exception* from a *hpx::error*.

Parameters

- *e*: The parameter *e* holds the *hpx::error* code the new exception should encapsulate.

exception (*std::system_error const &e*)

Construct a *hpx::exception* from a *boost::system_error*.

exception (*std::error_code const &e*)

Construct a *hpx::exception* from a *boost::system::error_code* (this is new for Boost V1.69). This constructor is required to compensate for the changes introduced as a resolution to LWG3162 (<https://cplusplus.github.io/LWG/issue3162>).

exception (*error e, char const *msg, throwmode mode = plain*)

Construct a *hpx::exception* from a *hpx::error* and an error message.

Parameters

- *e*: The parameter *e* holds the *hpx::error* code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *mode*: The parameter *mode* specifies whether the returned *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

exception (*error e, std::string const &msg, throwmode mode = plain*)

Construct a *hpx::exception* from a *hpx::error* and an error message.

Parameters

- *e*: The parameter *e* holds the *hpx::error* code the new exception should encapsulate.
- *msg*: The parameter *msg* holds the error message the new exception should encapsulate.
- *mode*: The parameter *mode* specifies whether the returned *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

~exception()

Destruct a *hpx::exception*

Exceptions

- nothing:

error get_error() const

The function *get_error()* returns the *hpx::error* code stored in the referenced instance of a *hpx::exception*. It returns the *hpx::error* code this exception instance was constructed from.

Exceptions

- nothing:

error_code get_error_code (throwmode mode = plain) const

The function *get_error_code()* returns a *hpx::error_code* which represents the same error condition as this *hpx::exception* instance.

Parameters

- mode: The parameter mode specifies whether the returned *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

```
struct thread_interrupted : public exception
```

```
#include <exception.hpp> A hpx::thread_interrupted is the exception type used by HPX to interrupt a running HPX thread.
```

The *hpx::thread_interrupted* type is the exception type used by HPX to interrupt a running thread.

A running thread can be interrupted by invoking the `interrupt()` member function of the corresponding *hpx::thread* object. When the interrupted thread next executes one of the specified interruption points (or if it is currently blocked whilst executing one) with interruption enabled, then a *hpx::thread_interrupted* exception will be thrown in the interrupted thread. If not caught, this will cause the execution of the interrupted thread to terminate. As with any other exception, the stack will be unwound, and destructors for objects of automatic storage duration will be executed.

If a thread wishes to avoid being interrupted, it can create an instance of *hpx::this_thread::disable_interruption*. Objects of this class disable interruption for the thread that created them on construction, and restore the interruption state to whatever it was before on destruction.

```
void f()
{
    // interruption enabled here
    {
        hpx::this_thread::disable_interruption di;
        // interruption disabled
        {
            hpx::this_thread::disable_interruption di2;
            // interruption still disabled
        } // di2 destroyed, interruption state restored
        // interruption still disabled
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

The effects of an instance of *hpx::this_thread::disable_interruption* can be temporarily reversed by constructing an instance of *hpx::this_thread::restore_interruption*, passing in the *hpx::this_thread::disable_interruption* object in question. This will restore the interruption state to what it was when the *hpx::this_thread::disable_interruption* object was constructed, and then disable interruption again when the *hpx::this_thread::restore_interruption* object is destroyed.

```
void g()
{
    // interruption enabled here
    {
        hpx::this_thread::disable_interruption di;
        // interruption disabled
        {
            hpx::this_thread::restore_interruption ri(di);
            // interruption now enabled
        } // ri destroyed, interruption disable again
    } // di destroyed, interruption state restored
}
```

(continues on next page)

(continued from previous page)

```
// interruption now enabled
}
```

At any point, the interruption state for the current thread can be queried by calling `hpx::this_thread::interruption_enabled()`.

```
namespace hpx
```

Enums

`enum throwmode`

Encode error category for new `error_code`.

Values:

```
plain = 0
rethrow = 1
lightweight = 0x80
```

Variables

`error_code throws`

Predefined `error_code` object used as “throw on error” tag.

The predefined `hpx::error_code` object `hpx::throws` is supplied for use as a “throw on error” tag.

Functions that specify an argument in the form ‘`error_code& ec=throws`’ (with appropriate namespace qualifiers), have the following error handling semantics:

If `&ec != &throws` and an error occurred: `ec.value()` returns the implementation specific error number for the particular error that occurred and `ec.category()` returns the `error_category` for `ec.value()`.

If `&ec != &throws` and an error did not occur, `ec.clear()`.

If an error occurs and `&ec == &throws`, the function throws an exception of type `hpx::exception` or of a type derived from it. The exception’s `get_errorcode()` member function returns a reference to an `hpx::error_code` object with the behavior as specified above.

Defines

```
HPX_DEFINE_ERROR_INFO (NAME, TYPE)
```

```
namespace hpx
```

Functions

```
template<typename E>HPX_NORETURN void hpx::throw_with_info(E && e, exception_info && info)
template<typename E>HPX_NORETURN void hpx::throw_with_info(E && e, exception_info const && info)

template<typename E>
exception_info *get_exception_info(E &e)

template<typename E>
exception_info const *get_exception_info(E const &e)

template<typename E, typename F>
auto invoke_with_exception_info(E const &e, F &&f)

template<typename F>
auto invoke_with_exception_info(std::exception_ptr const &p, F &&f)

template<typename F>
auto invoke_with_exception_info(hpx::error_code const &ec, F &&f)

template<typename Tag, typename Type>
struct error_info
```

Public Types

```
template<>
using tag = Tag

template<>
using type = Type
```

Public Functions

```
error_info(Type const &value)
error_info(Type &&value)
```

Public Members

Type _value

```
class exception_info
Subclassed by hpx::detail::exception_with_info_base
```

Public Functions

```
exception_info()
exception_info(exception_info const &other)
exception_info(exception_info &&other)
exception_info &operator=(exception_info const &other)
exception_info &operator=(exception_info &&other)
virtual ~exception_info()
template<typename ...ErrorInfo>
exception_info &set (ErrorInfo&&... tagged_values)
template<typename Tag>
Tag::type const *get () const
```

Private Types

```
using node_ptr = std::shared_ptr<detail::exception_info_node_base>
```

Private Members

```
node_ptr _data
namespace hpx
```

```
class exception_list : public hpx::exception
#include <exception_list.hpp> The class exception_list is a container of exception_ptr objects parallel algorithms may use to communicate uncaught exceptions encountered during parallel execution to the caller of the algorithm
```

The type *exception_list*::const_iterator fulfills the requirements of a forward iterator.

Public Types

```
using iterator=exception_list_type::const_iterator
bidirectional iterator
```

Public Functions

```
std::size_t size() const
```

The number of *exception_ptr* objects contained within the *exception_list*.

Note Complexity: Constant time.

```
exception_list_type::const_iterator begin() const
```

An iterator referring to the first *exception_ptr* object contained within the *exception_list*.

```
exception_list_type::const_iterator end() const
```

An iterator which is the past-the-end value for the *exception_list*.

Defines

HPX_THROW_EXCEPTION (*errcode, f, ...*)

Throw a *hpx::exception* initialized from the given parameters.

The macro *HPX_THROW_EXCEPTION* can be used to throw a *hpx::exception*. The purpose of this macro is to prepend the source file name and line number of the position where the exception is thrown to the error message. Moreover, this associates additional diagnostic information with the exception, such as file name and line number, locality id and thread id, and stack backtrace from the point where the exception was thrown.

The parameter *errcode* holds the *hpx::error* code the new exception should encapsulate. The parameter *f* is expected to hold the name of the function exception is thrown from and the parameter *msg* holds the error message the new exception should encapsulate.

```
void raise_exception()
{
    // Throw a hpx::exception initialized from the given parameters.
    // Additionally associate with this exception some detailed
    // diagnostic information about the throw-site.
    HPX_THROW_EXCEPTION(hpx::no_success, "raise_exception", "simulated error");
}
```

Example:

HPX_THROWS_IF (*ec, errcode, f, ...*)

Either throw a *hpx::exception* or initialize *hpx::error_code* from the given parameters.

The macro *HPX_THROWS_IF* can be used to either throw a *hpx::exception* or to initialize a *hpx::error_code* from the given parameters. If *&ec == &hpx::throws*, the semantics of this macro are equivalent to *HPX_THROW_EXCEPTION*. If *&ec != &hpx::throws*, the *hpx::error_code* instance *ec* is initialized instead.

The parameter *errcode* holds the *hpx::error* code from which the new exception should be initialized. The parameter *f* is expected to hold the name of the function exception is thrown from and the parameter *msg* holds the error message the new exception should encapsulate.

execution

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace hpx

namespace execution

namespace experimental

Variables

```
hpx::execution::experimental::bulk_t bulk
struct bulk_t : public hpx::functional::detail::tag_priority<bulk_t>
```

Friends

```
template<typename Sender, typename Shape, typename F>
friend constexpr auto tag_override_invoke(bulk_t, Sender &&sender, Shape
const &shape, F &&f)

template<typename Sender, typename Shape, typename F>
friend constexpr auto tag_fallback_invoke(bulk_t, Sender &&sender, Shape
const &shape, F &&f)

template<typename Sender, typename Shape, typename F>
friend constexpr auto tag_fallback_invoke(bulk_t, Sender &&sender, Shape
&&shape, F &&f)

template<typename Shape, typename F>
friend constexpr auto tag_fallback_invoke(bulk_t, Shape &&shape, F &&f)

namespace hpx
```

```
namespace execution
```

```
namespace experimental
```

Variables

```
hpx::execution::experimental::ensure_started_t ensure_started
struct ensure_started_t : public hpx::functional::detail::tag_fallback<ensure_started_t>
```

Friends

```
template<typename Sender, typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(ensure_started_t, Sender
&&sender, Allocator const
&allocator = {})

template<typename Sender, typename Allocator>
friend constexpr auto tag_fallback_invoke(ensure_started_t, detail::split_sender<Sender,
Allocator, detail::submission_type::eager>
sender, Allocator const& = {})

template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(ensure_started_t, Allocator const
&allocator = {})
```

```
namespace hpx
```

```
    namespace execution
```

```
        namespace experimental
```

Variables

```
hpx::execution::experimental::execute_t execute
```

```
struct execute_t : public hpx::functional::detail::tag_fallback<execute_t>
```

Friends

```
template<typename Scheduler, typename F>
friend constexpr auto tagFallbackInvoke(execute_t, Scheduler &&scheduler,
                                         F &&f)
```

```
namespace hpx
```

```
    namespace execution
```

```
        namespace experimental
```

Variables

```
constexpr struct hpx::execution::experimental::just_t just
```

```
struct just_t
```

Public Functions

```
template<typename ...Ts>
constexpr auto operator()(Ts&&... ts) const
```

```
namespace hpx
```

```
    namespace execution
```

```
        namespace experimental
```

Variables

```
constexpr struct hpx::execution::experimental::keep_future_t keep_future
struct keep_future_t
```

Public Functions

```
template<typename Future>
constexpr auto operator() (Future &&future) const

constexpr auto operator() () const

namespace hpx

namespace execution

namespace experimental
```

Variables

```
hpx::execution::experimental::let_error_t let_error
struct let_error_t : public hpx::functional::detail::tag_fallback<let_error_t>
```

Friends

```
template<typename PredecessorSender, typename F>
friend constexpr auto tagFallbackInvoke (let_error_t, PredecessorSender
                                         &&predecessor_sender, F &&f)

template<typename F>
friend constexpr auto tagFallbackInvoke (let_error_t, F &&f)

namespace hpx

namespace execution

namespace experimental
```

Variables

```
hpx::execution::experimental::let_value_t let_value
struct let_value_t : public hpx::functional::detail::tag_fallback<let_value_t>
```

Friends

```
template<typename PredecessorSender, typename F>
friend constexpr auto tagFallback_invoke(let_value_t, PredecessorSender
                                         &&predecessor_sender, F &&f)

template<typename F>
friend constexpr auto tagFallback_invoke(let_value_t, F &&f)

namespace hpx

namespace execution

namespace experimental
```

Variables

```
hpx::execution::experimental::make_future_t make_future

struct make_future_t : public hpx::functional::detail::tagFallback<make_future_t>
```

Friends

```
template<typename Sender, typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tagFallback_invoke(make_future_t, Sender &&sender,
                                         Allocator const &allocator = Allocator{ })

template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tagFallback_invoke(make_future_t, Allocator const
                                         &allocator = Allocator{ })

namespace hpx

namespace execution

namespace experimental
```

Variables

```
hpx::execution::experimental::schedule_from_t schedule_from

struct schedule_from_t : public hpx::functional::detail::tagFallback<schedule_from_t>
```

Friends

```
template<typename Scheduler, typename Sender>
friend constexpr auto tag_fallback_invoke(schedule_from_t, Scheduler
    &&scheduler, Sender &&prede-
    cessor_sender)

namespace hpx

namespace execution

namespace experimental
```

Variables

```
hpx::execution::experimental::split_t split

struct split_t : public hpx::functional::detail::tag_fallback<split_t>
```

Friends

```
template<typename Sender, typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(split_t, Sender &&sender, Allocator const &allocator = {})

template<typename Sender, typename Allocator>
friend constexpr auto tag_fallback_invoke(split_t, detail::split_sender<Sender,
    Allocator, detail::submission_type::lazy>
    sender, Allocator const& = {})

template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(split_t, Allocator const &allocat-
    or = {})

namespace hpx
```

namespace execution

namespace experimental

Variables

```
hpx::execution::experimental::start_detached_t start_detached

struct start_detached_t : public hpx::functional::detail::tag_fallback<start_detached_t>
```

Friends

```
template<typename Sender, typename Allocator = hpx::util::internal_allocator<>>
friend constexpr void tag_fallback_invoke(start_detached_t, Sender
                                         &&sender, Allocator const
                                         &allocator = Allocator{})
```

```
template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(start_detached_t, Allocator const
                                         &allocator = Allocator{})
```

```
namespace hpx
```

```
namespace execution
```

```
namespace experimental
```

Variables

```
hpx::execution::experimental::sync_wait_t sync_wait
```

```
struct sync_wait_t : public hpx::functional::detail::tag_fallback<sync_wait_t>
```

Friends

```
template<typename Sender>
friend constexpr auto tag_fallback_invoke(sync_wait_t, Sender &&sender)
```

```
friend constexpr auto tag_fallback_invoke(sync_wait_t)
```

```
namespace hpx
```

```
namespace execution
```

```
namespace experimental
```

Variables

```
hpx::execution::experimental::then_t then
```

```
struct then_t : public hpx::functional::detail::tag_fallback<then_t>
```

Friends

```
template<typename Sender, typename F>
friend constexpr auto tag_fallback_invoke(then_t, Sender &&sender, F &&f)

template<typename F>
friend constexpr auto tag_fallback_invoke(then_t, F &&f)

namespace hpx

namespace execution

namespace experimental
```

Variables

```
hpx::execution::experimental::transfer_t transfer

struct transfer_t : public hpx::functional::detail::tag_priority<transfer_t>
```

Friends

```
template<typename Sender, typename Scheduler>
friend constexpr auto tag_override_invoke(transfer_t, Sender &&sender,
                                         Scheduler &&scheduler)

template<typename Sender, typename Scheduler>
friend constexpr auto tag_fallback_invoke(transfer_t, Sender &&predeces-
                                             sor_sender, Scheduler &&sched-
                                             uler)

template<typename Scheduler>
friend constexpr auto tag_fallback_invoke(transfer_t, Scheduler &&sched-
                                             uler)

namespace hpx
```

namespace execution

namespace experimental

Variables

```
hpx::execution::experimental::transfer_just_t transfer_just

struct transfer_just_t : public hpx::functional::detail::tag_fallback<transfer_just_t>
```

Friends

```
template<typename Scheduler, typename ...Ts>
friend constexpr auto tag_fallback_invoke(transfer_just_t, Scheduler
    &&scheduler, Ts&&... ts)
```

namespace hpx

```
namespace execution
```

```
namespace experimental
```

Variables

```
hpx::execution::experimental::when_all_t when_all
```

```
struct when_all_t : public hpx::functional::detail::tag_fallback<when_all_t>
```

Friends

```
template<typename ...Senders>
friend constexpr auto tag_fallback_invoke(when_all_t, Senders&&... senders)
```

namespace hpx

```
namespace execution
```

struct auto_chunk_size

```
#include <auto_chunk_size.hpp> Loop iterations are divided into pieces and then assigned to threads.  
The number of loop iterations combined is determined based on measurements of how long the execution of 1% of the overall number of iterations takes. This executor parameters type makes sure that as many loop iterations are combined as necessary to run for the amount of time specified.
```

Public Functions

```
constexpr auto_chunk_size(std::uint64_t num_iters_for_timing = 0)  
Construct an auto_chunk_size executor parameters object
```

Note Default constructed auto_chunk_size executor parameter types will use 80 microseconds as the minimal time for which any of the scheduled chunks should run.

```
auto_chunk_size(hpx::chrono::steady_duration const &rel_time, std::uint64_t  
    num_iters_for_timing = 0)  
Construct an auto_chunk_size executor parameters object
```

Parameters

- *rel_time*: [in] The time duration to use as the minimum to decide how many loop iterations should be combined.

```
namespace parallel
```

```
    namespace execution
```

Typedefs

```
typedef hpx::is_sequenced_execution_policy<T> instead
```

```
namespace hpx
```

```
    namespace execution
```

```
        struct dynamic_chunk_size
```

#include <dynamic_chunk_size.hpp> Loop iterations are divided into pieces of size *chunk_size* and then dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. If *chunk_size* is not specified, the default chunk size is 1.

Note This executor parameters type is equivalent to OpenMP's DYNAMIC scheduling directive.

Public Functions

```
constexpr dynamic_chunk_size(std::size_t chunk_size = 1)
```

Construct a dynamic_chunk_size executor parameters object

Parameters

- *chunk_size*: [in] The optional chunk size to use as the number of loop iterations to schedule together. The default chunk size is 1.

```
namespace hpx
```

```
    namespace parallel
```

```
        namespace execution
```

Variables

```
hpx::parallel::execution::has_pending_closures_t has_pending_closures
```

```
hpx::parallel::execution::get_pu_mask_t get_pu_mask
```

```
hpx::parallel::execution::set_scheduler_mode_t set_scheduler_mode
```

```
struct get_pu_mask_t : public hpx::functional::detail::tagFallback<get_pu_mask_t>
```

#include <execution_information.hpp> Retrieve the bitmask describing the processing units the given thread is allowed to run on

All threads::executors invoke sched.get_pu_mask().

Note If the executor does not support this operation, this call will always invoke `hpx::threads::get_pu_mask()`

Parameters

- `exec`: [in] The executor object to use for querying the number of pending tasks.
- `topo`: [in] The topology object to use to extract the requested information.
- `threan_num`: [in] The sequence number of the thread to retrieve information for.

Private Functions

```
template<typename Executor>
decltype(auto) friend tag_fallback_invoke(get_pu_mask_t, Executor&&,  
                                         threads::topology &topo, std::size_t  
                                         thread_num)

template<typename Executor>
decltype(auto) friend tag_invoke(get_pu_mask_t, Executor &&exec, threads::topology  
&topo, std::size_t thread_num)

struct has_pending_closures_t : public hpx::functional::detail::tag_fallback<has_pending_closures_t>  
#include <execution_information.hpp> Retrieve whether this executor has operations pending or  
not.
```

Note If the executor does not expose this information, this call will always return `false`

Parameters

- `exec`: [in] The executor object to use to extract the requested information for.

Private Functions

```
template<typename Executor>
decltype(auto) friend tag_fallback_invoke(has_pending_closures_t, Executor&&)  
                                         Executor&&)

template<typename Executor>
decltype(auto) friend tag_invoke(has_pending_closures_t, Executor &&exec)

struct set_scheduler_mode_t : public hpx::functional::detail::tag_fallback<set_scheduler_mode_t>  
#include <execution_information.hpp> Set various modes of operation on the scheduler under-  
neath the given executor.
```

Note This calls `exec.set_scheduler_mode(mode)` if it exists; otherwise it does nothing.

Parameters

- `exec`: [in] The executor object to use.
- `mode`: [in] The new mode for the scheduler to pick up

Friends

```
template<typename Executor, typename Mode>
void tag_fallback_invoke(set_scheduler_mode_t, Executor&&, Mode const&)

template<typename Executor, typename Mode>
void tag_invoke(set_scheduler_mode_t, Executor &&exec, Mode const &mode)

namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

Functions

```
template<typename ...Params>
constexpr executor_parameters_join<Params...>::type join_executor_parameters(Params&&...
    params)

template<typename Param>
constexpr Param &&join_executor_parameters(Param &&param)

template<typename ...Params>
struct executor_parameters_join
```

Public Types

```
template<>
using type = detail::executor_parameters<std::decay_t<Params>...>

template<typename Param>
struct executor_parameters_join<Param>
```

Public Types

```
template<>
using type = Param

namespace hpx

namespace parallel

namespace execution
```

Variables

```
hpx::parallel::execution::get_chunk_size_t get_chunk_size
hpx::parallel::execution::maximal_number_of_chunks_t maximal_number_of_chunks
hpx::parallel::execution::reset_thread_distribution_t reset_thread_distribution
hpx::parallel::execution::processing_units_count_t processing_units_count
hpx::parallel::execution::mark_begin_execution_t mark_begin_execution
hpx::parallel::execution::mark_end_of_scheduling_t mark_end_of_scheduling
hpx::parallel::execution::mark_end_execution_t mark_end_execution

struct get_chunk_size_t : public hpx::functional::detail::tag_fallback<get_chunk_size_t>
    #include <execution_parameters_fwd.hpp> Return the number of invocations of the given function f which should be combined into a single task
```

Note The parameter f is expected to be a nullary function returning a `std::size_t` representing the number of iteration the function has already executed (i.e. which don't have to be scheduled anymore).

Parameters

- `params`: [in] The executor parameters object to use for determining the chunk size for the given number of tasks `num_tasks`.
- `exec`: [in] The executor object which will be used for scheduling of the loop iterations.
- `f`: [in] The function which will be optionally scheduled using the given executor.
- `cores`: [in] The number of cores the number of chunks should be determined for.
- `num_tasks`: [in] The number of tasks the chunk size should be determined for

Private Functions

```
template<typename Parameters, typename Executor, typename F>
decltype(auto) friend tagFallback_invoke(get_chunk_size_t, Parameters
                                         &&params, Executor &&exec, F
                                         &&f, std::size_t cores, std::size_t
                                         num_tasks)
```



```
struct mark_begin_execution_t : public hpx::functional::detail::tag_fallback<mark_begin_execution_t>
    #include <execution_parameters_fwd.hpp> Mark the begin of a parallel algorithm execution
```

Note This calls `params.mark_begin_execution(exec)` if it exists; otherwise it does nothing.

Parameters

- `params`: [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
decltype(auto) friend tagFallback_invoke(mark_begin_execution_t, Parameters
                                         &&params, Executor &&exec)

struct mark_end_execution_t : public hpx::functional::detail::tagFallback<mark_end_execution_t>
#include <execution_parameters_fwd.hpp> Mark the end of a parallel algorithm execution
```

Note This calls params.mark_end_execution(exec) if it exists; otherwise it does nothing.

Parameters

- params: [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
decltype(auto) friend tagFallback_invoke(mark_end_execution_t, Parameters
                                         &&params, Executor &&exec)

struct mark_end_of_scheduling_t : public hpx::functional::detail::tagFallback<mark_end_of_scheduling_t>
#include <execution_parameters_fwd.hpp> Mark the end of scheduling tasks during parallel algorithm execution
```

Note This calls params.mark_begin_execution(exec) if it exists; otherwise it does nothing.

Parameters

- params: [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
decltype(auto) friend tagFallback_invoke(mark_end_of_scheduling_t, Parameters
                                         &&params, Executor &&exec)

struct maximal_number_of_chunks_t : public hpx::functional::detail::tagFallback<maximal_number_of_chunks_t>
#include <execution_parameters_fwd.hpp> Return the largest reasonable number of chunks to create for a single algorithm invocation.
```

Parameters

- params: [in] The executor parameters object to use for determining the number of chunks for the given number of *cores*.
- exec: [in] The executor object which will be used for scheduling of the loop iterations.
- cores: [in] The number of cores the number of chunks should be determined for.
- num_tasks: [in] The number of tasks the chunk size should be determined for

Private Functions

```
template<typename Parameters, typename Executor>
decltype(auto) friend tag_fallback_invoke(maximal_number_of_chunks_t,    Pa-
                                         rams &&params,    Executor
                                         &&exec, std::size_t cores, std::size_t
                                         num_tasks)

struct processing_units_count_t : public hpx::functional::detail::tag_fallback<processing_units_count_t>
#include <execution_parameters_fwd.hpp> Retrieve the number of (kernel-)threads used by the
associated executor.
```

Note This calls params.processing_units_count(Executor&&) if it exists; otherwise it forwards the request to the executor parameters object.

Parameters

- params: [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
decltype(auto) friend tag_fallback_invoke(processing_units_count_t, Parameters
                                         &&params, Executor &&exec)

struct reset_thread_distribution_t : public hpx::functional::detail::tag_fallback<reset_thread_distribution_t>
#include <execution_parameters_fwd.hpp> Reset the internal round robin thread distribution
scheme for the given executor.
```

Note This calls params.reset_thread_distribution(exec) if it exists; otherwise it does nothing.

Parameters

- params: [in] The executor parameters object to use for resetting the thread distribution scheme.
- exec: [in] The executor object to use.

Private Functions

```
template<typename Parameters, typename Executor>
decltype(auto) friend tag_fallback_invoke(reset_thread_distribution_t, Parameters
                                         &&params, Executor &&exec)

namespace hpx

namespace execution

struct guided_chunk_size
#include <guided_chunk_size.hpp> Iterations are dynamically assigned to threads in blocks as threads
request them until no blocks remain to be assigned. Similar to dynamic_chunk_size except that the
block size decreases each time a number of loop iterations is given to a thread. The size of the initial
block is proportional to number_of_iterations / number_of_cores. Subsequent blocks are proportional
to number_of_iterations_remaining / number_of_cores. The optional chunk size parameter defines the
minimum block size. The default chunk size is 1.
```

Note This executor parameters type is equivalent to OpenMP's GUIDED scheduling directive.

Public Functions

constexpr guided_chunk_size (*std::size_t min_chunk_size = 1*)

Construct a guided_chunk_size executor parameters object

Parameters

- *min_chunk_size*: [in] The optional minimal chunk size to use as the minimal number of loop iterations to schedule together. The default minimal chunk size is 1.

namespace hpx

namespace execution

struct persistent_auto_chunk_size

#include <persistent_auto_chunk_size.hpp> Loop iterations are divided into pieces and then assigned to threads. The number of loop iterations combined is determined based on measurements of how long the execution of 1% of the overall number of iterations takes. This executor parameters type makes sure that as many loop iterations are combined as necessary to run for the amount of time specified.

Public Functions

constexpr persistent_auto_chunk_size (*std::uint64_t num_iters_for_timing = 0*)

Construct an persistent_auto_chunk_size executor parameters object

Note Default constructed persistent_auto_chunk_size executor parameter types will use 0 microseconds as the execution time for each chunk and 80 microseconds as the minimal time for which any of the scheduled chunks should run.

persistent_auto_chunk_size (*hpx::chrono::steady_duration const &time_cs,*
std::uint64_t num_iters_for_timing = 0)

Construct an persistent_auto_chunk_size executor parameters object

Parameters

- *time_cs*: The execution time for each chunk.

persistent_auto_chunk_size (*hpx::chrono::steady_duration const &time_cs,*
hpx::chrono::steady_duration const &rel_time,
std::uint64_t num_iters_for_timing = 0)

Construct an persistent_auto_chunk_size executor parameters object

Parameters

- *rel_time*: [in] The time duration to use as the minimum to decide how many loop iterations should be combined.
- *time_cs*: The execution time for each chunk.

namespace hpx

```
namespace parallel

namespace execution

template<typename R, typename ...Ts>
class polymorphic_executor<R(Ts...)> : hpx::parallel::execution::detail::polymorphic_executor_base
    private
```

Public Types

```
template<typename T>
using future_type = hpx::future<R>
```

Public Functions

```
constexpr polymorphic_executor()
polymorphic_executor(polymorphic_executor const &other)
polymorphic_executor(polymorphic_executor &&other)
polymorphic_executor &operator=(polymorphic_executor const &other)
polymorphic_executor &operator=(polymorphic_executor &&other)

template<typename Exec, typename PE = typename std::decay<Exec>::type, typename Enable = typename
polymorphic_executor(Exec &&exec)

template<typename Exec, typename PE = typename std::decay<Exec>::type, typename Enable = typename
polymorphic_executor &operator=(Exec &&exec)

void reset()

template<typename F>
void post(F &&f, Ts... ts) const

template<typename F>
R sync_execute(F &&f, Ts... ts) const

template<typename F>
hpx::future<R> async_execute(F &&f, Ts... ts) const

template<typename F, typename Future>
hpx::future<R> then_execute(F &&f, Future &&predecessor, Ts&&... ts) const

template<typename F, typename Shape>
std::vector<R> bulk_sync_execute(F &&f, Shape const &s, Ts&&... ts) const

template<typename F, typename Shape>
std::vector<hpx::future<R>> bulk_async_execute(F &&f, Shape const &s, Ts&&... ts) const

template<typename F, typename Shape>
hpx::future<std::vector<R>> bulk_then_execute(F &&f, Shape const &s,
                                                hpx::shared_future<void> const
                                                &predecessor, Ts&&... ts) const
```

Private Types

```
template<>
using base_type = detail::polymorphic_executor_base

template<>
using vtable = detail::polymorphic_executor_vtable<R (Ts...) >
```

Private Functions

```
void assign (std::nullptr_t)

template<typename Exec>
void assign (Exec &&exec)
```

Private Static Functions

```
static constexpr vtable const *get_empty_vtable()

template<typename T>
static constexpr vtable const *get_vtable()

namespace hpx

namespace parallel

namespace execution
```

Variables

```
constexpr create_rebound_policy_t create_rebound_policy = { }

struct create_rebound_policy_t
```

Public Functions

```
template<typename ExPolicy, typename Executor, typename Parameters>
constexpr decltype(auto) operator() (ExPolicy&&, Executor &&exec, Parameters
&&parameters) const
```

```
template<typename ExPolicy, typename Executor, typename Parameters>
struct rebind_executor
#include <rebind_executor.hpp> Rebind the type of executor used by an execution policy. The
execution category of Executor shall not be weaker than that of ExecutionPolicy.
```

Public Types

```
template<>
using type = typename policy_type::template rebind::type
    The type of the rebound execution policy.

namespace hpx

namespace execution

struct static_chunk_size
    #include <static_chunk_size.hpp> Loop iterations are divided into pieces of size chunk_size and then assigned to threads. If chunk_size is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
```

Note This executor parameters type is equivalent to OpenMP's STATIC scheduling directive.

Public Functions

```
constexpr static_chunk_size()
    Construct a static_chunk_size executor parameters object
```

Note By default the number of loop iterations is determined from the number of available cores and the overall number of loop iterations to schedule.

```
constexpr static_chunk_size(std::size_t chunk_size)
    Construct a static_chunk_size executor parameters object
```

Parameters

- *chunk_size*: [in] The optional chunk size to use as the number of loop iterations to run on a single thread.

```
namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

TypeDefs

```
template<typename Executor, typename T, typename ...Ts>
using executor_future_t = typename executor_future<Executor, T, Ts...>::type

template<typename Executor>
struct executor_context
```

Public Types

```
template<>
using type = std::decay_t<decltype(std::declval<Executorconst&>().context())>

template<typename Executor>
struct executor_execution_category
```

Public Types

```
template<>
using type = hpx::util::detected_or_t<hpx::execution::unsequenced_execution_tag, execution_category, Executor>
```

Private Types

```
template<typename T>
using execution_category = typename T::execution_category

template<typename Executor>
struct executor_index
```

Public Types

```
template<>
using type = hpx::util::detected_or_t<typename executor_shape<Executor>::type, index_type, Executor>
```

Private Types

```
template<typename T>
using index_type = typename T::index_type

template<typename Executor>
struct executor_parameters_type
```

Public Types

```
template<>
using type = hpx::util::detected_or_t<hpx::execution::static_chunk_size, parameters_type, Executor>
```

Private Types

```
template<typename T>
using parameters_type = typename T::parameters_type

template<typename Executor>
struct executor_shape
```

Public Types

```
template<>
using type = hpx::util::detected_or_t<std::size_t, shape_type, Executor>
```

Private Types

```
template<typename T>
using shape_type = typename T::shape_type
```

```
namespace traits
```

Typedefs

```
template<typename Executor>
using executor_context_t = typename executor_context<Executor>::type

template<typename Executor>
using executor_execution_category_t = typename executor_execution_category<Executor>::type

template<typename Executor>
using executor_shape_t = typename executor_shape<Executor>::type

template<typename Executor>
using executor_index_t = typename executor_index<Executor>::type

template<typename Executor, typename T, typename ...Ts>
using executor_future_t = typename executor_future<Executor, T, Ts...>::type

template<typename Executor>
using executor_parameters_type_t = typename executor_parameters_type<Executor>::type
```

Variables

```
template<typename T>
constexpr bool has_post_member_v = has_post_member<T>::value

template<typename T>
constexpr bool has_sync_execute_member_v = has_sync_execute_member<T>::value

template<typename T>
constexpr bool has_async_execute_member_v = has_async_execute_member<T>::value

template<typename T>
constexpr bool has_then_execute_member_v = has_then_execute_member<T>::value

template<typename T>
constexpr bool has_bulk_sync_execute_member_v = has_bulk_sync_execute_member<T>::value

template<typename T>
constexpr bool has_bulk_async_execute_member_v = has_bulk_async_execute_member<T>::value

template<typename T>
constexpr bool has_bulk_then_execute_member_v = has_bulk_then_execute_member<T>::value

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename Executor, typename Future, typename F>
using future_then_executor_result_t = typename future_then_executor_result<Executor, Future, F>::type

namespace hpx
```

Variables

```
template<typename T>
constexpr bool is_execution_policy_v = is_execution_policy<T>::value

template<typename T>
constexpr bool is_parallel_execution_policy_v = is_parallel_execution_policy<T>::value

template<typename T>
constexpr bool is_sequenced_execution_policy_v = is_sequenced_execution_policy<T>::value

template<typename T>
constexpr bool is_async_execution_policy_v = is_async_execution_policy<T>::value

template<typename T>
struct is_async_execution_policy : public hpx::detail::is_async_execution_policy<std::decay<T>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy makes algorithms
asynchronous
```

1. The type *is_async_execution_policy* can be used to detect asynchronous execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.
2. If *T* is the type of a standard or implementation-defined execution policy, *is_async_execution_policy*<*T*> shall be publicly derived from *integral_constant*<bool, true>, otherwise from *integral_constant*<bool, false>.
3. The behavior of a program that adds specializations for *is_async_execution_policy* is undefined.

```
template<typename T>
struct is_execution_policy : public hpx::detail::is_execution_policy<std::decay<T>::type>
#include <is_execution_policy.hpp>

1. The type is_execution_policy can be used to detect execution policies for the purpose of excluding 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 hpx::detail::is_parallel_execution_policy<std::decay<T>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy enables parallelization
```

1. The type *is_parallel_execution_policy* can be used to detect parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

2. If `T` is the type of a standard or implementation-defined execution policy, `is_parallel_execution_policy<T>` shall be publicly derived from `integral_constant<bool, true>`, otherwise from `integral_constant<bool, false>`.

3. The behavior of a program that adds specializations for `is_parallel_execution_policy` is undefined.

```
template<typename T>
struct is_sequenced_execution_policy : public hpx::detail::is_sequenced_execution_policy<std::decay<T>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy does not enable parallelization
```

1. The type `is_sequenced_execution_policy` can be used to detect non-parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.
2. If `T` is the type of a standard or implementation-defined execution policy, `is_sequenced_execution_policy<T>` shall be publicly derived from `integral_constant<bool, true>`, otherwise from `integral_constant<bool, false>`.
3. The behavior of a program that adds specializations for `is_sequenced_execution_policy` is undefined.

```
namespace hpx
```

```
namespace parallel
```

```
namespace traits
```

Functions

```
std::size_t count_bits(bool value)
```

Functions

```
void check_exception_ptr(std::exception_ptr eptr)
struct void_sender
```

Public Types

```
template<template<typename...> class Tuple, template<typename...> class Variant>
using value_types = Variant<Tuple<>>

template<template<typename...> class Variant>
using error_types = Variant<>
```

Public Static Attributes

```
constexpr bool sends_done = false
```

Friends

```
template<typename R>
operation_state<R> tag_invoke(hpx::execution::experimental::connect_t, void_sender, R &&r)
```

```
template<typename R>
struct operation_state
```

Public Members

```
std::decay_t<R> r
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)
```

```
template<typename ...Ts>
struct error_sender
```

Public Types

```
template<template<typename...> class Tuple, template<typename...> class Variant>
using value_types = Variant<Tuple<Ts...>>
```

```
template<template<typename...> class Variant>
using error_types = Variant<std::exception_ptr>
```

Public Static Attributes

```
constexpr bool sends_done = false
```

Friends

```
template<typename R>
operation_state<R> tag_invoke(hpx::execution::experimental::connect_t, error_sender, R &&r)
```

```
template<typename R>
struct operation_state
```

Public Members

```
template<>
std::decay_t<R> r
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)

template<typename F>
struct callback_receiver
```

Public Members

```
std::decay_t<F> f
std::atomic<bool> &set_value_called
```

Friends

```
template<typename E>
void tag_invoke(hpx::execution::experimental::set_error_t, callback_receiver&&, E &&)

void tag_invoke(hpx::execution::experimental::set_done_t, callback_receiver&&)

template<typename ...Ts>
auto tag_invoke(hpx::execution::experimental::set_value_t, callback_receiver &&r, Ts&&... ts)

template<typename F>
struct error_callback_receiver
```

Public Members

```
std::decay_t<F> f
std::atomic<bool> &set_error_called
bool expect_set_value = false
```

Friends

```
template<typename E>
void tag_invoke(hpx::execution::experimental::set_error_t, error_callback_receiver &&r, E &&e)

void tag_invoke(hpx::execution::experimental::set_done_t, error_callback_receiver&&)

template<typename ...Ts>
void tag_invoke(hpx::execution::experimental::set_value_t, error_callback_receiver &&r, Ts&&...)
```

template<typename F>

```
struct void_callback_helper
```

Public Functions

```
void operator() () const  
template<typename T>  
void operator() (T &&t)
```

Public Members

```
std::decay_t<F> f  
template<typename T>  
struct error_typed_sender
```

Public Types

```
template<template<class...> class Tuple, template<class...> class Variant>  
using value_types = Variant<Tuple<T>>  
template<template<class...> class Variant>  
using error_types = Variant<std::exception_ptr>
```

Public Static Attributes

```
constexpr bool sends_done = false
```

Friends

```
template<typename R>  
auto tag_invoke(hpx::execution::experimental::connect_t, error_typed_sender&&, R &&r)  
template<typename R>  
struct operation_state
```

Public Members

```
template<>  
std::decay_t<R> r
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)  
struct custom_sender_tag_invoke
```

Public Types

```
template<template<typename...> class Tuple, template<typename...> class Variantusing value_types = Variant<Tuple<>>

template<template<typename...> class Variantusing error_types = Variant<>
```

Public Functions

```
template<typename R>
operation_state<R> connect (R &&r)
```

Public Members

```
std::atomic<bool> &tag_invoke_overload_called
```

Public Static Attributes

```
constexpr bool sends_done = false

template<typename R>
struct operation_state
```

Public Functions

```
void start ()
```

Public Members

```
std::decay_t<R> r

struct custom_sender
Subclassed by custom_sender2
```

Public Types

```
template<template<class...> class Tuple, template<class...> class Variantusing value_types = Variant<Tuple<>>

template<template<class...> class Variantusing error_types = Variant<std::exception_ptr>
```

Public Members

```
std::atomic<bool> &start_called
std::atomic<bool> &connect_called
std::atomic<bool> &tag_invoke_overload_called
```

Public Static Attributes

```
constexpr bool sends_done = false
```

Friends

```
template<typename R>
auto tag_invoke(hpx::execution::experimental::connect_t, custom_sender &&s, R &&r)

template<typename R>
struct operation_state
```

Public Members

```
std::atomic<bool> &start_called
std::decay_t<R> r
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)

template<typename T>
struct custom_typed_sender
```

Public Types

```
template<template<class...> class Tuple, template<class...> class Variant>
using value_types = Variant<Tuple<>>

template<template<class...> class Variant>
using error_types = Variant<std::exception_ptr>
```

Public Members

```
std::decay_t<T> x
std::atomic<bool> &start_called
std::atomic<bool> &connect_called
std::atomic<bool> &tag_invoke_overload_called
```

Public Static Attributes

```
constexpr bool sends_done = false
```

Friends

```
template<typename R>
auto tag_invoke(hpx::execution::experimental::connect_t, custom_typed_sender &&s, R &&r)

template<typename R>
struct operation_state
```

Public Members

```
template<>
std::decay_t<T> x

template<>
std::atomic<bool> &start_called

template<>
std::decay_t<R> r
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)

struct custom_sender2 : public custom_sender
```

Public Functions

```
custom_sender2(custom_sender s)
```

```
template<typename T>
struct custom_type
```

Public Members

```
std::atomic<bool> &tag_invoke_overload_called
std::decay_t<T> x

struct custom_type_non_default_constructible
```

Public Functions

```
custom_type_non_default_constructible()
custom_type_non_default_constructible(int x)
custom_type_non_default_constructible(custom_type_non_default_constructible&&)
custom_type_non_default_constructible &operator=(custom_type_non_default_constructible&&)
custom_type_non_default_constructible(custom_type_non_default_constructible
                                     const&)
custom_type_non_default_constructible &operator=(custom_type_non_default_constructible
                                     const&)
```

Public Members

int **x**

```
struct custom_type_non_default_constructible_non_copyable
```

Public Functions

```
custom_type_non_default_constructible_non_copyable()
custom_type_non_default_constructible_non_copyable(int x)
custom_type_non_default_constructible_non_copyable(custom_type_non_default_constructible_non_copyable&
custom_type_non_default_constructible_non_copyable &operator=(custom_type_non_default_constructible_non_copyable&
custom_type_non_default_constructible_non_copyable(custom_type_non_default_constructible_non_copyable
                                     const&)
custom_type_non_default_constructible_non_copyable &operator=(custom_type_non_default_constructible_non_copyable
                                     const&)
```

Public Members

int **x**

```
struct scheduler
Subclassed by scheduler2
```

Public Functions

```
bool operator==(scheduler const&) const
bool operator!=(scheduler const&) const
```

Public Members

```
std::atomic<bool> &schedule_called  
std::atomic<bool> &execute_called  
std::atomic<bool> &tag_invoke_overload_called
```

Friends

```
template<typename F>  
void tag_invoke(hpx::execution::experimental::execute_t, scheduler s, F &&f)  
sender tag_invoke(hpx::execution::experimental::schedule_t, scheduler s)  
struct sender
```

Public Types

```
template<template<class...> class Tuple, template<class...> class Variant>  
using value_types = Variant<Tuple<>>;  
template<template<class...> class Variant>  
using error_types = Variant<std::exception_ptr>;
```

Public Static Attributes

```
constexpr bool sends_done = false
```

Friends

```
template<typename R>  
auto tag_invoke(hpx::execution::experimental::connect_t, sender&&, R &&r)  
template<typename R>  
struct operation_state
```

Public Members

```
std::decay_t<R> r
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)  
struct scheduler2 : public scheduler
```

Public Functions

```
scheduler2 (scheduler s)
```

execution_base

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace execution_base
```

```
struct agent_base
```

Public Functions

```
virtual ~agent_base()

virtual std::string description() const = 0

virtual context_base const &context() const = 0

virtual void yield(char const *desc) = 0

virtual void yield_k(std::size_t k, char const *desc) = 0

virtual void suspend(char const *desc) = 0

virtual void resume(char const *desc) = 0

virtual void abort(char const *desc) = 0

virtual void sleep_for(hpx::chrono::steady_duration const &sleep_duration, char
                      const *desc) = 0

virtual void sleep_until(hpx::chrono::steady_time_point const &sleep_time, char
                        const *desc) = 0
```

```
namespace hpx
```

```
namespace execution_base
```

```
class agent_ref
```

Public Functions

```
constexpr agent_ref()
constexpr agent_ref(agent_base *impl)
constexpr agent_ref(agent_ref const&)
constexpr agent_ref &operator=(agent_ref const&)
constexpr agent_ref (agent_ref&&)
constexpr agent_ref &operator=(agent_ref&&)
constexpr operator bool() const
void reset(agent_base *impl = nullptr)
void yield(char const *desc = "hpx::execution_base::agent_ref::yield")
void yield_k(std::size_t k, char const *desc = "hpx::execution_base::agent_ref::yield_k")
void suspend(char const *desc = "hpx::execution_base::agent_ref::suspend")
void resume(char const *desc = "hpx::execution_base::agent_ref::resume")
void abort(char const *desc = "hpx::execution_base::agent_ref::abort")
template<typename Rep, typename Period>
void sleep_for(std::chrono::duration<Rep, Period> const &sleep_duration, char const
              *desc = "hpx::execution_base::agent_ref::sleep_for")
template<typename Clock, typename Duration>
void sleep_until(std::chrono::time_point<Clock, Duration> const &sleep_time, char const
                 *desc = "hpx::execution_base::agent_ref::sleep_until")
agent_base &ref()
```

Private Functions

```
void sleep_for(hpx::chrono::steady_duration const &sleep_duration, char const *desc)
void sleep_until(hpx::chrono::steady_time_point const &sleep_time, char const
                 *desc)
```

Private Members

```
agent_base *impl_
```

Friends

```

friend constexpr bool operator== (agent_ref const &lhs, agent_ref const &rhs)
friend constexpr bool operator!= (agent_ref const &lhs, agent_ref const &rhs)
std::ostream &operator<< (std::ostream&, agent_ref const&)

namespace hpx

namespace execution

namespace experimental

template<typename ...Ts>
class any_sender : private hpx::execution::experimental::detail::any_sender_static_empty_vtable_helper<Ts...>

```

Public Types

```

template<template<typename...> class Tuple, template<typename...> class Variant>
using value_types = Variant<Tuple<Ts...>>;
template<template<typename...> class Variant>
using error_types = Variant<std::exception_ptr>;

```

Public Functions

```

any_sender()
template<typename Sender, typename = std::enable_if_t<!std::is_same_v<std::decay_t<Sender>, any_sender>>
any_sender(Sender &&sender)
template<typename Sender, typename = std::enable_if_t<!std::is_same_v<std::decay_t<Sender>, any_sender>>
any_sender &operator=(Sender &&sender)
~any_sender()
any_sender(any_sender&&)
any_sender(any_sender const&)
any_sender &operator=(any_sender&&)
any_sender &operator=(any_sender const&)

```

Public Static Attributes

```
constexpr bool sends_done = false
```

Private Types

```
template<>
using base_type = detail::any_sender_base<Ts...>

template<typename Sender>
using impl_type = detail::any_sender_impl<Sender, Ts...>

template<>
using storage_type = hpx::detail::copyable_sbo_storage<base_type, 4 * sizeof(void*)>
```

Private Members

```
storage_type storage = { }
```

Friends

```
template<typename R>
detail::any_operation_state tag_invoke(hpx::execution::experimental::connect_t,
                                         any_sender &s, R &&r)

template<typename R>
detail::any_operation_state tag_invoke(hpx::execution::experimental::connect_t,
                                         any_sender &&s, R &&r)

template<typename ...Ts>
class unique_any_sender : private hpx::execution::experimental::detail::any_sender_static_empty_vtable_
```

Public Types

```
template<template<typename...> class Tuple, template<typename...> class Variantusing value_types = Variant<Tuple<Ts...>>

template<template<typename...> class Variant>
using error_types = Variant<std::exception_ptr>
```

Public Functions

```
unique_any_sender()
```

```
template<typename Sender, typename = std::enable_if_t<!std::is_same_v<std::decay_t<Sender>, unique_any_sender>>
unique_any_sender(Sender &&sender)
```

```
template<typename Sender, typename = std::enable_if_t<!std::is_same_v<std::decay_t<Sender>, unique_any_sender>>
unique_any_sender &operator=(Sender &&sender)
```

```
~unique_any_sender()
```

```
unique_any_sender(unique_any_sender&&)
```

```
unique_any_sender (unique_any_sender const&)
unique_any_sender &operator= (unique_any_sender&&)
unique_any_sender &operator= (unique_any_sender const&)
```

Public Static Attributes

```
constexpr bool sends_done = false
```

Private Types

```
template<>
using base_type = detail::unique_any_sender_base<Ts...>
template<typename Sender>
using impl_type = detail::unique_any_sender_impl<Sender, Ts...>
template<>
using storage_type = hpx::detail::movable_sbo_storage<base_type, 4 * sizeof(void*)>
```

Private Members

```
storage_type storage = {}
```

Friends

```
template<typename R>
detail::any_operation_state tag_invoke (hpx::execution::experimental::connect_t,
                                         unique_any_sender &&s, R &&r)
```

```
namespace hpx
```

```
namespace execution
```

```
namespace experimental
```

Variables

```
template<typename Scheduler>
constexpr get_completion_scheduler_t<Scheduler> get_completion_scheduler = {}

namespace hpx
```

```
namespace execution_base
```

```
struct context_base
```

Public Functions

```
virtual ~context_base()

virtual resource_base const &resource() const = 0

namespace hpx

namespace execution

namespace experimental
```

Functions

```
template<typename O>
void start(O &&o)
    start is a customization point object. The expression
    hpx::execution::experimental::start(r) is equivalent to:
    • r.start(), if that expression is valid. If the function selected does not signal the receiver
        r's done channel, the program is ill-formed (no diagnostic required).
    • Otherwise, ^start(r), if that expression is valid, with overload resolution performed in a context
        that include the declaration void start();
    • Otherwise, the expression is ill-formed.
```

The customization is implemented in terms of `hpx::functional::tag_invoke`.

Variables

```
hpx::execution::experimental::start_t start

template<typename O>
constexpr bool is_operation_state_v = is_operation_state<O>::value

template<typename O>
struct is_operation_state
    #include <operation_state.hpp> An operation_state is an object rep-
    resenting the asynchronous operation that has been returned from calling
    hpx::execution::experimental::connect with a sender and a receiver.
    The only operation on an operation_state is:
    • hpx::execution::experimental::start
    hpx::execution::experimental::start can be called exactly once. Once it
    has been invoked, the caller needs to ensure that the receiver's completion signaling
    operations strongly happen before the destructor of the state is called. The call to
    hpx::execution::experimental::start needs to happen strongly before the com-
    pletion signaling operations.

namespace hpx

namespace execution

namespace experimental
```

Functions

```
template<typename R, typename ...As>
void set_value (R &&r, As&&... as)
    set_value is a customization point object. The expression
    hpx::execution::set_value(r, as...) is equivalent to:
    • r.set_value(as...), if that expression is valid. If the function selected does not send
        the value(s) as... to the Receiver r's value channel, the program is ill-formed (no diagnostic
        required).
    • Otherwise, `set_value(r, as...), if that expression is valid, with overload resolution performed
        in a context that include the declaration void set_value();
    • Otherwise, the expression is ill-formed.
The customization is implemented in terms of hpx::functional::tag_invoke.
```

```
template<typename R>
void set_done (R &&r)
    set_done is a customization point object. The expression hpx::execution::set_done(r)
    is equivalent to:
    • r.set_done(), if that expression is valid. If the function selected does not signal the Re-
        ceiver r's done channel, the program is ill-formed (no diagnostic required).
    • Otherwise, `set_done(r), if that expression is valid, with overload resolution performed in a
        context that include the declaration void set_done();
    • Otherwise, the expression is ill-formed.
The customization is implemented in terms of hpx::functional::tag_invoke.
```

```
template<typename R, typename E>
void set_error (R &&r, E &&e)
    set_error is a customization point object. The expression
    hpx::execution::set_error(r, e) is equivalent to:
    • r.set_error(e), if that expression is valid. If the function selected does not send the error
        e the Receiver r's error channel, the program is ill-formed (no diagnostic required).
    • Otherwise, `set_error(r, e), if that expression is valid, with overload resolution performed in a
        context that include the declaration void set_error();
    • Otherwise, the expression is ill-formed.
The customization is implemented in terms of hpx::functional::tag_invoke.
```

Variables

```
hpx::execution::experimental::set_value_t set_value
hpx::execution::experimental::set_error_t set_error
hpx::execution::experimental::set_done_t set_done
template<typename T, typename E = std::exception_ptr>
constexpr bool is_receiver_v = is_receiver<T, E>::value

template<typename T, typename ...As>
constexpr bool is_receiver_of_v = is_receiver_of<T, As...>::value

template<typename T, typename ...As>
constexpr bool is_nothrow_receiver_of_v = is_nothrow_receiver_of<T, As...>::value

template<typename T, typename E>
struct is_receiver
#include <receiver.hpp> Receiving values from asynchronous computations is handled by the
```

Receiver concept. A Receiver needs to be able to receive an error or be marked as being canceled. As such, the Receiver concept is defined by having the following two customization points defined, which form the completion-signal operations:

- `hpx::execution::experimental::set_done`
- `hpx::execution::experimental::set_error`

Those two functions denote the completion-signal operations. The Receiver contract is as follows:

- None of a Receiver's completion-signal operation shall be invoked before `hpx::execution::experimental::start` has been called on the operation state object that was returned by connecting a Receiver to a sender `hpx::execution::experimental::connect`.
- Once `hpx::execution::start` has been called on the operation state object, exactly one of the Receiver's completion-signal operation shall complete without an exception before the Receiver is destroyed

Once one of the Receiver's completion-signal operation has been completed without throwing an exception, the Receiver contract has been satisfied. In other words: The asynchronous operation has been completed.

See `hpx::execution::experimental::is_receiver_of`

```
template<typename T, typename ...As>
struct is_receiver_of
    #include <receiver.hpp> The receiver_of concept is a refinement of the Receiver concept by requiring one additional completion-signal operation:
```

- `hpx::execution::set_value`

This completion-signal operation adds the following to the Receiver's contract:

- If `hpx::execution::set_value` exits with an exception, it is still valid to call `hpx::execution::set_error` or `hpx::execution::set_done`

See `hpx::execution::traits::is_receiver`

```
namespace hpx
```

```
namespace execution_base
```

```
    struct resource_base
```

```
        #include <resource_base.hpp> TODO: implement, this is currently just a dummy.
```

Public Functions

```
    virtual ~resource_base()
```

```
namespace hpx
```

```
namespace execution
```

```
    namespace experimental
```

TypeDefs

```
template<typename S, typename R>
using connect_result_t = typename hpx::util::invoke_result<connect_t, S, R>::type
```

Functions

```
template<typename S, typename R>
void connect(S &&s, R &&r)
```

connect is a customization point object. For some subexpression *s* and *r*, let *S* be the type such that `decltype((s))` is *S* and let *R* be the type such that `decltype((r))` is *R*. The result of the expression `hpx::execution::experimental::connect(s, r)` is then equivalent to:

- `s.connect(r)`, if that expression is valid and returns a type satisfying the `operation_state`
See `hpx::execution::experimental::traits::is_operation_state` and if *S* satisfies the `sender` concept.
- `s.connect(r)`, if that expression is valid and returns a type satisfying the `operation_state`
See `hpx::execution::experimental::traits::is_operation_state` and if *S* satisfies the `sender` concept. Overload resolution is performed in a context that include the declaration `void connect();`
- Otherwise, the expression is ill-formed.

The customization is implemented in terms of `hpx::functional::tag_invoke`.

Variables

```
template<typename Sender>
constexpr bool is_sender_v = is_sender<Sender>::value

hpx::execution::experimental::connect_t connect

hpx::execution::experimental::schedule_t schedule

template<typename Scheduler>
constexpr bool is_scheduler_v = is_scheduler<Scheduler>::value

struct invocable_archetype
```

Public Functions

```
void operator()()
```

```
template<typename Sender>
struct is_sender
```

#include <sender.hpp> The name `schedule` denotes a customization point object. For some subexpression *s*, let *S* be `decltype((s))`. The expression `schedule(s)` is expression-equivalent to:

- * `s.schedule()`, **if** that expression is valid and its type models `sender`.
 - * Otherwise, `schedule(s)`, **if** that expression is valid and its type models `sender` with overload resolution performed in a context that includes the declaration

(continues on next page)

(continued from previous page)

```
void schedule();  
  
and that does not include a declaration of schedule.  
  
* Otherwise, schedule(s) is ill-formed.
```

The customization is implemented in terms of `hpx::functional::tag_invoke`. A sender is a type that is describing an asynchronous operation. The operation itself might not have started yet. In order to get the result of this asynchronous operation, a sender needs to be connected to a receiver with the corresponding value, error and done channels:

- `hpx::execution::experimental::connect`

In addition, `hpx::execution::experimental:::sender_traits` needs to be specialized in some form.

A sender's destructor shall not block pending completion of submitted operations.

```
template<typename Sender, typename Receiver>  
struct is_sender_to  
    #include <sender.hpp>  
See is_sender  
  
template<typename Sender>  
struct sender_traits  
    #include <sender.hpp> sender_traits expose the different value and error types exposed  
    by a sender. This can be either specialized directly for user defined sender types or embedded  
    value_types, error_types and sends_done inside the sender type can be provided.  
  
Subclassed      by      hpx::execution::experimental:::sender_traits<      Sender  
&      >,      hpx::execution::experimental:::sender_traits<      Sender      &&  
>,      hpx::execution::experimental:::sender_traits<      Sender      const      >,  
hpx::execution::experimental:::sender_traits< Sender volatile >  
  
template<>  
struct sender_traits<void>
```

Public Types

```
template<>  
using __unspecialized = void  
  
namespace hpx  
  
namespace execution_base  
  
namespace this_thread
```

Functions

```

hpx::execution_base::agent_ref agent ()

void yield (char const *desc = "hpx::execution_base::this_thread::yield")

void yield_k (std::size_t k, char const *desc = "hpx::execution_base::this_thread::yield_k")

void suspend (char const *desc = "hpx::execution_base::this_thread::suspend")

template<typename Rep, typename Period>
void sleep_for (std::chrono::duration<Rep, Period> const &sleep_duration, char const
               *desc = "hpx::execution_base::this_thread::sleep_for")

template<class Clock, class Duration>
void sleep_until (std::chrono::time_point<Clock, Duration> const &sleep_time, char const
                  *desc = "hpx::execution_base::this_thread::sleep_for")

struct reset_agent

```

Public Functions

```

reset_agent (detail::agent_storage*, agent_base &impl)
reset_agent (agent_base &impl)
~reset_agent ()

```

Public Members

```

detail::agent_storage *storage_
agent_base *old_

namespace util

```

Functions

```

template<typename Predicate>
void yield_while (Predicate &&predicate, const char *thread_name = nullptr, bool allow_timed_suspension = true)

namespace hpx

```

```
namespace traits
```

Typedefs

```
template<typename T>
using is_one_way_executor_t = typename is_one_way_executor<T>::type

template<typename T>
using is_never_blocking_one_way_executor_t = typename is_never_blocking_one_way_executor<T>::type

template<typename T>
using is_bulk_one_way_executor_t = typename is_bulk_one_way_executor<T>::type

template<typename T>
using is_two_way_executor_t = typename is_two_way_executor<T>::type

template<typename T>
using is_bulk_two_way_executor_t = typename is_bulk_two_way_executor<T>::type

template<typename T>
using is_executor_any_t = typename is_executor_any<T>::type
```

Variables

```
template<typename T>
constexpr bool is_one_way_executor_v = is_one_way_executor<T>::value

template<typename T>
constexpr bool is_never_blocking_one_way_executor_v = is_never_blocking_one_way_executor<T>::value

template<typename T>
constexpr bool is_bulk_one_way_executor_v = is_bulk_one_way_executor<T>::value

template<typename T>
constexpr bool is_two_way_executor_v = is_two_way_executor<T>::value

template<typename T>
constexpr bool is_bulk_two_way_executor_v = is_bulk_two_way_executor<T>::value

template<typename T>
constexpr bool is_executor_any_v = is_executor_any<T>::value

template<typename Executor>
struct extract_executor_parameters<Executor, typename hpx::util::always_void<typename Executor::executor_params>
```

Public Types

```
template<>
using type = typename Executor::executor_parameters_type

template<typename Parameters>
struct extract_has_variable_chunk_size<Parameters, typename hpx::util::always_void<typename Parameters::has>
```

Public Types

```
template<>
using type = typename Parameters::has_variable_chunk_size

namespace hpx

namespace parallel

namespace execution
```

Typedefs

```
template<typename T>
using is_executor_parameters_t = typename is_executor_parameters<T>::type
```

Variables

```
template<typename T>
constexpr bool is_executor_parameters_v = is_executor_parameters<T>::value

template<typename Executor, typename Enable = void>
struct extract_executor_parameters
```

Public Types

```
template<>
using type = sequential_executor_parameters

template<typename Executor>
struct extract_executor_parameters<Executor, typename hpx::util::always_void<typename Executor>
```

Public Types

```
template<>
using type = typename Executor::executor_parameters_type

template<typename Parameters, typename Enable = void>
struct extract_has_variable_chunk_size
```

Public Types

```
template<>
using type = std::false_type

template<typename Parameters>
struct extract_has_variable_chunk_size<Parameters, typename hpx::util::always_void<typename
```

Public Types

```
template<>
using type = typename Parameters::has_variable_chunk_size

namespace traits
```

TypeDefs

```
template<typename T>
using is_executor_parameters_t = typename is_executor_parameters<T>::type
```

Variables

```
template<typename T>
constexpr bool is_executor_parameters_v = is_executor_parameters<T>::value
```

executors

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace execution
```

```
namespace experimental
```

Functions

```
template<typename Executor>
constexpr auto tag_fallback_invoke(with_annotation_t, Executor &&exec, char
                                         const *annotation)

template<typename Executor>
auto tag_fallback_invoke(with_annotation_t, Executor &&exec, std::string annotation)

template<typename BaseExecutor>
struct annotating_executor
    #include <annotating_executor.hpp> A annotating_executor wraps any other executor and adds
    the capability to add annotations to the launched threads.
```

Public Functions

```
template<typename Executor, typename Enable = std::enable_if_t<hpx::traits::is_executor_any_v<Executor>>
constexpr annotating_executor(Executor &exec, char const *annotation =
                           nullptr)

template<typename Executor, typename Enable = std::enable_if_t<hpx::traits::is_executor_any_v<Executor>>
annotating_executor(Executor &&exec, std::string annotation)

namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

TypeDefs

```
using current_executor = parallel::execution::thread_pool_executor

namespace this_thread
```

Functions

`parallel::execution::current_executor get_executor(error_code &ec = throws)`

Returns a reference to the executor which was used to create the current thread.

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
namespace threads
```

Functions

`parallel::execution::current_executor get_executor(thread_id_type const &id, error_code &ec = throws)`

Returns a reference to the executor which was used to create the given thread.

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
namespace hpx
```

```
    namespace execution
```

Variables

```
constexpr task_policy_tag task = {}
```

Default sequential execution policy object.

```
constexpr non_task_policy_tag non_task = {}
```

```
constexpr sequenced_policy seq = {}
```

Default sequential execution policy object.

```
constexpr parallel_policy par = {}
```

Default parallel execution policy object.

```
constexpr parallel_unsequenced_policy par_unseq = {}
```

Default vector execution policy object.

```
constexpr unsequenced_policy unseq = {}
```

Default vector execution policy object.

```
struct 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.

Public Types

```
using executor_type = parallel_executor
```

The type of the executor associated with this execution policy.

```
using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type
```

The type of the associated executor parameters object which is associated with this execution policy

```
using execution_category = parallel_execution_tag
```

The category of the execution agents created by this execution policy.

Public Functions

```
constexpr parallel_task_policy operator() (task_policy_tag) const
```

Create a new *parallel_policy* referencing a chunk size.

Return The new *parallel_policy*

Parameters

- *tag*: [in] Specify that the corresponding asynchronous execution policy should be used

```
constexpr decltype(auto) operator() (non_task_policy_tag) const
```

Create a new *parallel_policy* from itself

Return The new *parallel_policy*

Parameters

- **tag:** [in] Specify that the corresponding asynchronous execution policy should be used

```
template<typename Executor>
constexpr decltype(auto) on (Executor &&exec) const
    Create a new parallel_policy referencing an executor and a chunk size.
```

Return The new *parallel_policy*

Parameters

- **exec:** [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with

```
template<typename ...Parameters>
constexpr decltype(auto) with (Parameters&&... params) const
    Create a new parallel_policy from the given execution parameters
```

Note Requires: `is_executor_parameters<Parameters>::value` is true

Return The new *parallel_policy*

Template Parameters

- **Parameters:** The type of the executor parameters to associate with this execution policy.

Parameters

- **params:** [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

executor_type &**executor** ()

Return the associated executor object.

constexpr *executor_type* **const** &**executor** () **const**

Return the associated executor object.

executor_parameters_type &**parameters** ()

Return the associated executor parameters object.

constexpr *executor_parameters_type* **const** &**parameters** () **const**

Return the associated executor parameters object.

Private Functions

```
template<typename Archive>
constexpr void serialize (Archive&, const unsigned int)
```

Private Members

executor_type **exec_**

executor_parameters_type **params_**

Friends

```
friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access

template<typename Executor_, typename Parameters_>
struct rebind
    #include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The
    execution category of Executor shall not be weaker than that of this execution policy
```

Public Types

```
template<>
using type = parallel_policy_shim<Executor_, Parameters_>
The type of the rebound execution policy.

template<typename Executor, typename Parameters>
struct parallel_policy_shim
    #include <execution_policy.hpp> The class parallel_policy_shim is an execution policy type used as
    a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's
    execution may be parallelized.
```

Public Types

```
template<>
using executor_type = std::decay_t<Executor>
The type of the executor associated with this execution policy.

template<>
using executor_parameters_type = std::decay_t<Parameters>
The type of the associated executor parameters object which is associated with this execution
policy

template<>
using execution_category = typename hpx::traits::executor_execution_category<executor_type>::type
The category of the execution agents created by this execution policy.
```

Public Functions

```
constexpr parallel_task_policy_shim<Executor, Parameters> operator() (task_policy_tag)
    const
Create a new parallel_policy referencing a chunk size.
```

Return The new *parallel_policy*

Parameters

- tag: [in] Specify that the corresponding asynchronous execution policy should be used

```
constexpr decltype(auto) operator() (non_task_policy_tag) const
Create a new parallel_policy from itself
```

Return The new *parallel_policy*

Parameters

- `tag`: [in] Specify that the corresponding asynchronous execution policy should be used

```
template<typename Executor_>
constexpr decltype(auto) on(Executor_ &&exec) const
    Create a new parallel_policy from the given executor
```

Note Requires: `is_executor<Executor>::value` is true

Return The new *parallel_policy*

Template Parameters

- `Executor`: The type of the executor to associate with this execution policy.

Parameters

- `exec`: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

```
template<typename ...Parameters_>
constexpr decltype(auto) with(Parameters_ &&... params) const
    Create a new parallel_policy_shim from the given execution parameters
```

Note Requires: `is_executor_parameters<Parameters>::value` is true

Return The new *parallel_policy_shim*

Template Parameters

- `Parameters`: The type of the executor parameters to associate with this execution policy.

Parameters

- `params`: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

```
executor_type &executor ()
    Return the associated executor object.
```

```
constexpr executor_type const &executor () const
    Return the associated executor object.
```

```
executor_parameters_type &parameters ()
    Return the associated executor parameters object.
```

```
constexpr executor_parameters_type const &parameters () const
    Return the associated executor parameters object.
```

```
template<typename Executor_, typename Parameters_>
struct rebind

```

`#include <execution_policy.hpp>` Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

Public Types

```
template<>
template<>
using type = parallel_policy_shim<Executor_, Parameters_>
    The type of the rebound execution policy.
```

```
struct parallel_task_policy
```

`#include <execution_policy.hpp>` Extension: The class *parallel_task_policy* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

The algorithm returns a future representing the result of the corresponding algorithm when invoked with the *parallel_policy*.

Public Types

using executor_type = parallel_executor

The type of the executor associated with this execution policy.

using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type

The type of the associated executor parameters object which is associated with this execution policy

using execution_category = parallel_execution_tag

The category of the execution agents created by this execution policy.

Public Functions

constexpr parallel_task_policy operator() (task_policy_tag) const

Create a new *parallel_task_policy* from itself

Return The new *parallel_task_policy*

Parameters

- *tag*: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) operator() (non_task_policy_tag) const

Create a new non task parallel policy from itself

Return The new non task *parallel_policy*

Parameters

- *tag*: [in] Specify that the corresponding execution policy should be used

template<typename Executor>

constexpr decltype(auto) on (Executor &&exec) const

Create a new *parallel_task_policy* from given executor

Note Requires: *is_executor<Executor>::value* is true

Return The new *parallel_task_policy*

Template Parameters

- *Executor*: The type of the executor to associate with this execution policy.

Parameters

- *exec*: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

template<typename ...Parameters>

constexpr decltype(auto) with (Parameters&&... params) const

Create a new *parallel_policy_shim* from the given execution parameters

Note Requires: all parameters are *executor_parameters*, different parameter types can't be duplicated

Return The new *parallel_policy_shim*

Template Parameters

- *Parameters*: The type of the executor parameters to associate with this execution policy.

Parameters

- `params`: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

`executor_type &executor()`

Return the associated executor object.

`constexpr executor_type const &executor() const`

Return the associated executor object.

`executor_parameters_type ¶meters()`

Return the associated executor parameters object.

`constexpr executor_parameters_type const ¶meters() const`

Return the associated executor parameters object.

Private Functions

```
template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)
```

Private Members

`executor_type exec_`

`executor_parameters_type params_`

Friends

```
friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access
```

```
template<typename Executor_, typename Parameters_>
struct rebind
```

`#include <execution_policy.hpp>` Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

Public Types

`template<>`

`using type = parallel_task_policy_shim<Executor_, Parameters_>`

The type of the rebound execution policy.

```
template<typename Executor, typename Parameters>
```

`struct parallel_task_policy_shim`

`#include <execution_policy.hpp>` Extension: The class `parallel_task_policy_shim` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading based on combining a underlying `parallel_task_policy` and an executor and indicate that a parallel algorithm's execution may be parallelized.

Public Types

```
template<>
using executor_type = std::decay_t<Executor>
    The type of the executor associated with this execution policy.

template<>
using executor_parameters_type = std::decay_t<Parameters>
    The type of the associated executor parameters object which is associated with this execution
    policy

template<>
using execution_category = typename hpx::traits::executor_execution_category<executor_type>::type
    The category of the execution agents created by this execution policy.
```

Public Functions

```
constexpr parallel_task_policy_shim operator() (task_policy_tag) const
    Create a new parallel_task_policy_shim from itself
```

Return The new *sequenced_task_policy*

Parameters

- *tag*: [in] Specify that the corresponding asynchronous execution policy should be used

```
constexpr decltype(auto) operator() (non_task_policy_tag) const
    Create a new non task parallel policy from itself
```

Return The new non task *parallel_policy_shim*

Parameters

- *tag*: [in] Specify that the corresponding execution policy should be used

```
template<typename Executor>
constexpr decltype(auto) on (Executor_ &&exec) const
    Create a new parallel_task_policy from the given executor
```

Note Requires: *is_executor<Executor>*::value is true

Return The new *parallel_task_policy*

Template Parameters

- *Executor*: The type of the executor to associate with this execution policy.

Parameters

- *exec*: [in] The executor to use for the execution of the parallel algorithm the returned
 execution policy is used with.

```
template<typename ...Parameters>
constexpr decltype(auto) with (Parameters_ &&... params) const
    Create a new parallel_policy_shim from the given execution parameters
```

Note Requires: all parameters are *executor_parameters*, different parameter types can't be duplicated

Return The new *parallel_policy_shim*

Template Parameters

- *Parameters*: The type of the executor parameters to associate with this execution policy.

Parameters

- **params:** [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

`executor_type &executor()`
 Return the associated executor object.

`constexpr executor_type const &executor() const`
 Return the associated executor object.

`executor_parameters_type ¶meters()`
 Return the associated executor parameters object.

`constexpr executor_parameters_type const ¶meters() const`
 Return the associated executor parameters object.

`template<typename Executor_, typename Parameters_>`
`struct rebind`
`#include <execution_policy.hpp>` Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

Public Types

`template<>`
`template<>`
`using type = parallel_task_policy_shim<Executor_, Parameters_>`
 The type of the rebound execution policy.

`struct parallel_unsequenced_policy`
`#include <execution_policy.hpp>` The class `parallel_unsequenced_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized.

Public Types

`using executor_type = parallel_executor`
 The type of the executor associated with this execution policy.

`using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type`
 The type of the associated executor parameters object which is associated with this execution policy

`using execution_category = parallel_execution_tag`
 The category of the execution agents created by this execution policy.

Public Functions

`constexpr decltype(auto) operator()(non_task_policy_tag) const`
 Create a new non task policy from itself

Return The non task parallel unsequenced policy

`executor_type &executor()`
 Return the associated executor object.

```
constexpr executor_type const &executor() const
    Return the associated executor object.

executor_parameters_type &parameters()
    Return the associated executor parameters object.

constexpr executor_parameters_type const &parameters() const
    Return the associated executor parameters object.
```

Private Functions

```
template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)
```

Private Members

```
executor_type exec_
executor_parameters_type params_
```

Friends

```
friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access

struct sequenced_policy
#include <execution_policy.hpp> The class sequenced_policy is an execution policy type used as a
unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's
execution may not be parallelized.
```

Public Types

```
using executor_type = sequenced_executor
    The type of the executor associated with this execution policy.

using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type
    The type of the associated executor parameters object which is associated with this execution
    policy

using execution_category = sequenced_execution_tag
    The category of the execution agents created by this execution policy.
```

Public Functions

```
constexpr sequenced_task_policy operator()(task_policy_tag) const
    Create a new sequenced_task_policy.
```

Return The new *sequenced_task_policy*

Parameters

- **tag:** [in] Specify that the corresponding asynchronous execution policy should be used

```
constexpr decltype(auto) operator() (non_task_policy_tag) const
Create a new sequenced_policy from itself.
```

Return The new *sequenced_policy*

```
template<typename Executor>
constexpr decltype(auto) on(Executor &&exec) const
Create a new sequenced_policy from the given executor
```

Note Requires: `is_executor<Executor>::value` is true

Return The new *sequenced_policy*

Template Parameters

- `Executor`: The type of the executor to associate with this execution policy.

Parameters

- `exec`: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

```
template<typename ...Parameters>
constexpr decltype(auto) with(Parameters&&... params) const
Create a new sequenced_policy from the given execution parameters
```

Note Requires: all parameters are `executor_parameters`, different parameter types can't be duplicated

Return The new *sequenced_policy*

Template Parameters

- `Parameters`: The type of the executor parameters to associate with this execution policy.

Parameters

- `params`: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

```
executor_type &executor()
```

Return the associated executor object.

```
constexpr executor_type const &executor() const
```

Return the associated executor object.

```
executor_parameters_type &parameters()
```

Return the associated executor parameters object.

```
constexpr executor_parameters_type const &parameters() const
```

Return the associated executor parameters object.

Private Functions

```
template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)
```

Private Members

```
executor_type exec_
executor_parameters_type params_
```

Friends

```
friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access

template<typename Executor_, typename Parameters_>
struct rebind
#include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The
execution category of Executor shall not be weaker than that of this execution policy
```

Public Types

```
template<>
using type = sequenced_policy_shim<Executor_, Parameters_>
The type of the rebound execution policy.

template<typename Executor, typename Parameters>
struct sequenced_policy_shim
#include <execution_policy.hpp> The class sequenced_policy is an execution policy type used as a
unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's
execution may not be parallelized.
```

Public Types

```
template<>
using executor_type = std::decay_t<Executor>
The type of the executor associated with this execution policy.

template<>
using executor_parameters_type = std::decay_t<Parameters>
The type of the associated executor parameters object which is associated with this execution
policy

template<>
using execution_category = typename hpx::traits::executor_execution_category<executor_type>::type
The category of the execution agents created by this execution policy.
```

Public Functions

constexpr sequenced_task_policy_shim<Executor, Parameters> **operator()** (task_policy_tag) **const**
 Create a new *sequenced_task_policy*.

Return The new *sequenced_task_policy_shim*

Parameters

- tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) **operator()** (non_task_policy_tag) **const**
 Create a new *sequenced_policy* from itself.

Return The new *sequenced_policy_shim*

template<typename **Executor**_>
constexpr decltype(auto) **on** (*Executor_* &&exec) **const**
 Create a new *sequenced_policy* from the given executor

Note Requires: *is_executor<Executor>::value* is true

Return The new *sequenced_policy*

Template Parameters

- Executor: The type of the executor to associate with this execution policy.

Parameters

- exec: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

template<typename ...**Parameters**_>
constexpr decltype(auto) **with** (*Parameters_* &&... params) **const**
 Create a new *sequenced_policy_shim* from the given execution parameters

Note Requires: all parameters are *executor_parameters*, different parameter types can't be duplicated

Return The new *sequenced_policy_shim*

Template Parameters

- Parameters: The type of the executor parameters to associate with this execution policy.

Parameters

- params: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

executor_type &executor()

Return the associated executor object.

constexpr **executor_type const &executor()** **const**

Return the associated executor object.

executor_parameters_type ¶meters()

Return the associated executor parameters object.

constexpr **executor_parameters_type const ¶meters()** **const**

Return the associated executor parameters object.

template<typename **Executor**_, typename **Parameters**_>

struct rebind

#include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

Public Types

template<>

template<>

using type = sequenced_policy_shim<Executor_, Parameters_>

The type of the rebound execution policy.

struct sequenced_task_policy

#include <execution_policy.hpp> Extension: The class *sequenced_task_policy* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may not be parallelized (has to run sequentially).

The algorithm returns a future representing the result of the corresponding algorithm when invoked with the *sequenced_policy*.

Public Types

using executor_type = sequenced_executor

The type of the executor associated with this execution policy.

using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type

The type of the associated executor parameters object which is associated with this execution policy

using execution_category = sequenced_execution_tag

The category of the execution agents created by this execution policy.

Public Functions

constexpr sequenced_task_policy **operator()** (task_policy_tag) **const**

Create a new *sequenced_task_policy* from itself

Return The new *sequenced_task_policy*

Parameters

- tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) **operator()** (non_task_policy_tag) **const**

Create a corresponding non task policy for this task policy

Return The non task sequential policy

template<typename Executor>

constexpr decltype(auto) **on** (Executor &&exec) **const**

Create a new *sequenced_task_policy* from the given executor

Note Requires: is_executor<Executor>::value is true

Return The new *sequenced_task_policy*

Template Parameters

- **Executor**: The type of the executor to associate with this execution policy.

Parameters

- **exec**: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

```
template<typename ...Parameters>
constexpr decltype(auto) with(Parameters&&... params) const
    Create a new sequenced_task_policy from the given execution parameters
```

Note Requires: all parameters are `executor_parameters`, different parameter types can't be duplicated

Return The new *sequenced_task_policy*

Template Parameters

- **Parameters**: The type of the executor parameters to associate with this execution policy.

Parameters

- **params**: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

```
executor_type &executor()
```

Return the associated executor object.

```
constexpr executor_type const &executor() const
```

Return the associated executor object.

```
executor_parameters_type &parameters()
```

Return the associated executor parameters object.

```
constexpr executor_parameters_type const &parameters() const
```

Return the associated executor parameters object.

Private Functions

```
template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)
```

Private Members

```
executor_type exec_
```

```
executor_parameters_type params_
```

Friends

```
friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access
```

```
template<typename Executor_, typename Parameters_>
struct rebind
```

#include <*execution_policy.hpp*> Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

Public Types

```
template<>
using type = sequenced_task_policy_shim<Executor_, Parameters_>
    The type of the rebound execution policy.
```

```
template<typename Executor, typename Parameters>
struct sequenced_task_policy_shim
#include <execution_policy.hpp> Extension: The class sequenced_task_policy_shim is an execution policy type used as a unique type to disambiguate parallel algorithm overloading based on combining a underlying sequenced_task_policy and an executor and indicate that a parallel algorithm's execution may not be parallelized (has to run sequentially).
```

The algorithm returns a future representing the result of the corresponding algorithm when invoked with the *sequenced_policy*.

Public Types

```
template<>
using executor_type = std::decay_t<Executor>
    The type of the executor associated with this execution policy.
```

```
template<>
using executor_parameters_type = std::decay_t<Parameters>
    The type of the associated executor parameters object which is associated with this execution policy
```

```
typedef hpx::traits::executor_execution_category<executor_type>::type execution_category
    The category of the execution agents created by this execution policy.
```

Public Functions

```
constexpr sequenced_task_policy_shim const &operator() (task_policy_tag) const
    Create a new sequenced_task_policy from itself
```

Return The new *sequenced_task_policy*
Parameters

- **tag:** [in] Specify that the corresponding asynchronous execution policy should be used

```
constexpr decltype(auto) operator() (non_task_policy_tag) const
    Create a corresponding non task policy for this task policy
```

Return The non task sequential shim policy

```
template<typename Executor_>
constexpr decltype(auto) on(Executor_ &&exec) const
    Create a new sequenced_task_policy from the given executor
```

Note Requires: `is_executor<Executor>::value` is true

Return The new *sequenced_task_policy*

Template Parameters

- **Executor:** The type of the executor to associate with this execution policy.

Parameters

- `exec`: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

```
template<typename ...Parameters_>
constexpr decltype(auto) with(Parameters_... params) const
    Create a new sequenced_task_policy_shim from the given execution parameters
```

Note Requires: all parameters are `executor_parameters`, different parameter types can't be duplicated

Return The new *sequenced_task_policy_shim*

Template Parameters

- `Parameters`: The type of the executor parameters to associate with this execution policy.

Parameters

- `params`: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

```
executor_type &executor()
```

Return the associated executor object.

```
constexpr executor_type const &executor() const
```

Return the associated executor object.

```
executor_parameters_type &parameters()
```

Return the associated executor parameters object.

```
constexpr executor_parameters_type const &parameters() const
```

Return the associated executor parameters object.

```
template<typename Executor_, typename Parameters_>
```

```
struct rebind
```

#include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

Public Types

```
template<>
```

```
template<>
```

```
using type = sequenced_task_policy_shim<Executor_, Parameters_>
```

The type of the rebound execution policy.

```
struct unsequenced_policy
```

#include <execution_policy.hpp> The class *unsequenced_policy* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized.

Public Types

using executor_type = sequenced_executor

The type of the executor associated with this execution policy.

using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type

The type of the associated executor parameters object which is associated with this execution policy

using execution_category = sequenced_execution_tag

The category of the execution agents created by this execution policy.

Public Functions

constexpr decltype(auto) operator() (non_task_policy_tag) const

Create a new non task policy from itself

Return The non task unsequenced policy

executor_type &executor()

Return the associated executor object.

constexpr executor_type const &executor() const

Return the associated executor object.

executor_parameters_type ¶meters()

Return the associated executor parameters object.

constexpr executor_parameters_type const ¶meters() const

Return the associated executor parameters object.

Private Functions

template<typename **Archive**>

constexpr void serialize (Archive&, const unsigned int)

Private Members

executor_type exec_

executor_parameters_type params_

Friends

friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t

friend hpx::execution::hpx::serialization::access

namespace parallel

namespace execution

TypeDefs

```
typedef hpx::execution::sequenced_executor instead
namespace hpx

namespace execution

namespace experimental
```

Functions

```
template<typename ExPolicy>
constexpr decltype(auto) tag_invoke(hpx::execution::experimental::with_annotation_t,
                                         ExPolicy &&policy, char const *annotation)

template<typename ExPolicy>
decltype(auto) tag_invoke(hpx::execution::experimental::with_annotation_t,           ExPolicy
                           &&policy, std::string annotation)

template<typename ExPolicy>
constexpr decltype(auto) tag_invoke(hpx::execution::experimental::get_annotation_t,
                                         ExPolicy &&policy)
```

namespace hpx

namespace execution

namespace experimental

Functions

```
std::ostream &operator<<(std::ostream &os, fork_join_executor::loop_schedule const
                           &schedule)

class fork_join_executor
#include <fork_join_executor.hpp> An executor with fork-join (blocking) semantics.
```

The *fork_join_executor* creates on construction a set of worker threads that are kept alive for the duration of the executor. Copying the executor has reference semantics, i.e. copies of a *fork_join_executor* hold a reference to the worker threads of the original instance. Scheduling work through the executor concurrently from different threads is undefined behaviour.

The executor keeps a set of worker threads alive for the lifetime of the executor, meaning other work will not be executed while the executor is busy or waiting for work. The executor has a customizable delay after which it will yield to other work. Since starting and resuming the worker threads is a slow operation the executor should be reused whenever possible for multiple adjacent parallel algorithms or invocations of bulk_(a)sync_execute.

Public Types

`enum loop_schedule`

Type of loop schedule for use with the `fork_join_executor`. `loop_schedule::static_` implies no work-stealing; `loop_schedule::dynamic` allows stealing when a worker has finished its local work.

Values:

`static_`
`dynamic`

Public Functions

```
fork_join_executor(threads::thread_priority priority = threads::thread_priority::high,
                   threads::thread_stacksize stacksize =
                     threads::thread_stacksize::small_, loop_schedule schedule =
                     loop_schedule::static_, std::chrono::nanoseconds yield_delay =
                     std::chrono::milliseconds(1))
```

Construct a `fork_join_executor`.

Parameters

- `priority`: The priority of the worker threads.
- `stacksize`: The stacksize of the worker threads. Must not be nostack.
- `schedule`: The loop schedule of the parallel regions.
- `yield_delay`: The time after which the executor yields to other work if it hasn't received any new work for bulk execution.

Defines

```
GUIDED_POOL_EXECUTOR_DEBUG
```

```
namespace hpx
```

Functions

```
static hpx::debug::enable_print<GUIDED_POOL_EXECUTOR_DEBUG> hpx::gpx_deb("GP_EXEC")
```

```
namespace parallel
```

```
namespace execution
```

```
template<typename Hint>
struct executor_execution_category<guided_pool_executor<Hint>>
```

Public Types

```
typedef hpx::execution::parallel_execution_tag type

template<typename Hint>
struct executor_execution_category<guided_pool_executor_shim<Hint>>
```

Public Types

```
typedef hpx::execution::parallel_execution_tag type

template<typename Tag>
struct guided_pool_executor<pool numa hint<Tag>>
```

Public Functions

```
guided_pool_executor(threads::thread_pool_base *pool, bool hp_sync = false)

guided_pool_executor(threads::thread_pool_base *pool, threads::thread_stacksize
stacksize, bool hp_sync = false)

guided_pool_executor(threads::thread_pool_base *pool, threads::thread_priority
priority, threads::thread_stacksize stacksize =
threads::thread_stacksize::default_, bool hp_sync = false)
```

```
template<typename F, typename ...Ts>
future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute(F
&&f,
Ts&&...
ts)
```

```
template<typename F, typename Future, typename ...Ts, typename = std::enable_if_t<hpx::traits::is_future<Future>>
auto then_execute(F &&f, Future &&predecessor, Ts&&... ts)
```

```
template<typename F, template<typename> class OuterFuture, typename ...InnerFutures, typename ...Ts>
auto then_execute(F &&f, OuterFuture<hpx::tuple<InnerFutures...>> &&predecessor,
Ts&&... ts)
```

```
template<typename F, typename ...InnerFutures, typename = std::enable_if_t<hpx::traits::is_future_tuple<hp
auto async_execute(F &&f, hpx::tuple<InnerFutures...> &&predecessor)
```

Private Members

```
threads::thread_pool_base *pool_
threads::thread_priority priority_
threads::thread_stacksize stacksize_
pool numa hint<Tag> hint_
bool hp_sync_
```

Friends

```
friend hpx::parallel::execution::guided_pool_executor_shim  
template<typename H>  
struct guided_pool_executor_shim
```

Public Functions

```
guided_pool_executor_shim(bool guided, threads::thread_pool_base *pool, bool  
hp_sync = false)
```

```
guided_pool_executor_shim(bool guided, threads::thread_pool_base *pool,  
threads::thread_stacksize stacksize, bool hp_sync =  
false)
```

```
guided_pool_executor_shim(bool guided, threads::thread_pool_base  
*pool, threads::thread_priority priority,  
threads::thread_stacksize stacksize =  
threads::thread_stacksize::default_, bool hp_sync  
= false)
```

```
template<typename F, typename ...Ts>  
future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute(F  
&&f,  
Ts&&...  
ts)
```

```
template<typename F, typename Future, typename ...Ts, typename = std::enable_if_t<hpx::traits::is_future<Future>>  
auto then_execute(F &&f, Future &&predecessor, Ts&&... ts)
```

Public Members

```
bool guided_  
guided_pool_executor<H> guided_exec_  
namespace hpx
```

```
namespace execution
```

```
namespace experimental
```

Typedefs

```
using print_on = hpx::debug::enable_print<false>
```

Functions

```
static constexpr print_on hpx::execution::experimental::lim_debug("LIMEXEC")
template<typename BaseExecutor>
struct limiting_executor
```

Public Types

```
template<>
using execution_category = typename BaseExecutor::execution_category
template<>
using executor_parameters_type = typename BaseExecutor::executor_parameters_type
```

Public Functions

```
limiting_executor(BaseExecutor &ex, std::size_t lower, std::size_t upper, bool block_on_destruction = true)
limiting_executor(std::size_t lower, std::size_t upper, bool block_on_destruction = true)
~limiting_executor()

limiting_executor const &context() const

template<typename F, typename ...Ts>
decltype(auto) sync_execute(F &&f, Ts&&... ts) const

template<typename F, typename ...Ts>
decltype(auto) async_execute(F &&f, Ts&&... ts)

template<typename F, typename Future, typename ...Ts>
decltype(auto) then_execute(F &&f, Future &&predecessor, Ts&&... ts)

template<typename F, typename ...Ts>
void post(F &&f, Ts&&... ts)

template<typename F, typename S, typename ...Ts>
decltype(auto) bulk_async_execute(F &&f, S const &shape, Ts&&... ts)

template<typename F, typename S, typename Future, typename ...Ts>
decltype(auto) bulk_then_execute(F &&f, S const &shape, Future &&predecessor,
                                Ts&&... ts)

void wait()
void wait_all()

void set_threshold(std::size_t lower, std::size_t upper)
```

Private Functions

```
void count_up()  
void count_down() const  
void set_and_wait(std::size_t lower, std::size_t upper)
```

Private Members

```
BaseExecutor executor_  
std::atomic<std::size_t> count_  
std::size_t lower_threshold_  
std::size_t upper_threshold_  
bool block_  
struct on_exit
```

Public Functions

```
template<>  
on_exit(limiting_executor const &this_e)  
  
template<>  
~on_exit()
```

Public Members

```
template<>  
limiting_executor const &executor_  
  
template<typename F, typename B = BaseExecutor, typename Enable = void>  
struct throttling_wrapper
```

Public Functions

```
template<>  
throttling_wrapper(limiting_executor &lim, BaseExecutor const&, F &&f)  
  
template<typename ...Ts>  
decltype(auto) operator()(Ts&&... ts)  
  
template<>  
bool exceeds_upper()  
  
template<>  
bool exceeds_lower()
```

Public Members

```
template<>
limiting_executor &limiting_<br/>
template<>
F f_<br/>
```

namespace hpx

namespace execution

TypeDefs

```
using parallel_executor = parallel_policy_executor<hpx::launch><br/>
template<typename Policy>
struct parallel_policy_executor<br/>
    #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

```
template<>
using execution_category = parallel_execution_tag<br/>
    Associate the parallel_execution_tag executor tag type as a default with this executor.<br/>
template<>
using executor_parameters_type = static_chunk_size<br/>
    Associate the static_chunk_size executor parameters type as a default with this executor.
```

Public Functions

```
constexpr parallel_policy_executor(threads::thread_priority priority,
                                    threads::thread_stacksize stacksize =
                                    threads::thread_stacksize::default_,
                                    threads::thread_schedule_hint sched-
                                    ulehint = {}, Policy l = paral-
                                    lel::execution::detail::get_default_policy<Policy>::call(),
                                    std::size_t hierarchical_threshold = hierar-
                                    chical_threshold_default_)
```

Create a new parallel executor.

```
constexpr parallel_policy_executor(threads::thread_stacksize stacksize,
                                    threads::thread_schedule_hint sched-
                                    ulehint = {}, Policy l = paral-
                                    lel::execution::detail::get_default_policy<Policy>::call())
```

```
constexpr parallel_policy_executor(threads::thread_schedule_hint
                                    schedulehint, Policy l = paral-
                                    lel::execution::detail::get_default_policy<Policy>::call())
```

```
constexpr parallel_policy_executor(Policy l = parallel::execution::detail::get_default_policy<Policy>::call())

constexpr parallel_policy_executor(thread::thread_pool_base *pool,
                                    thread::thread_priority priority =
                                        thread::thread_priority::default_,
                                    thread::thread_stacksize stacksize =
                                        thread::thread_stacksize::default_,
                                    thread::thread_schedule_hint sched-
                                    ulehint = {}, Policy l = parallel::execution::detail::get_default_policy<Policy>::call(),
                                    std::size_t hierarchical_threshold = hierar-
                                    chical_threshold_default_)
```

Friends

```
friend constexpr parallel_policy_executor tag_invoke(hpx::execution::experimental::with_hint_t,
                                                    parallel_policy_executor
                                                    const &exec,
                                                    hpx::threads::thread_schedule_hint
                                                    hint)

friend constexpr hpx::threads::thread_schedule_hint tag_invoke(hpx::execution::experimental::get_hint_t,
                                                               parallel_policy_executor
                                                               const &exec)

friend constexpr parallel_policy_executor tag_invoke(hpx::execution::experimental::with_priority_t,
                                                    parallel_policy_executor
                                                    const &exec,
                                                    hpx::threads::thread_priority
                                                    priority)

friend constexpr hpx::threads::thread_priority tag_invoke(hpx::execution::experimental::get_priority_t,
                                                       parallel_policy_executor
                                                       const &exec)

friend constexpr parallel_policy_executor tag_invoke(hpx::execution::experimental::with_annotation_t,
                                                    parallel_policy_executor
                                                    const &exec, char const
                                                    *annotation)

parallel_policy_executor tag_invoke(hpx::execution::experimental::with_annotation_t, parallel_policy_executor const &exec, std::string annotation)

friend constexpr char const *tag_invoke(hpx::execution::experimental::get_annotation_t,
                                       parallel_policy_executor const &exec)

template<>
struct parallel_policy_executor_aggregated<hpx::launch>
```

Public Types

```
template<>
using execution_category = hpx::execution::parallel_execution_tag
    Associate the parallel_execution_tag executor tag type as a default with this executor.

template<>
using executor_parameters_type = hpx::execution::static_chunk_size
    Associate the static_chunk_size executor parameters type as a default with this executor.
```

Public Functions

```
constexpr parallel_policy_executor_aggregated(hpx::launch          l      = hpx::launch::async_policy{},
                                                std::size_t spread = 4, std::size_t tasks = std::size_t(-1))
```

Create a new parallel executor.

```
template<typename F, typename S, typename ...Ts>
std::vector<hpx::future<void>> bulk_async_execute(F &&f, S const &shape, Ts&&... ts)
                                         const
```

namespace hpx

namespace parallel

namespace execution

Typedefs

```
using parallel_executor_aggregated = parallel_policy_executor_aggregated<hpx::launch::async_policy>
```

```
template<typename Policy = hpx::launch::async_policy>
struct parallel_policy_executor_aggregated
```

#include <parallel_executor_aggregated.hpp> A *parallel_executor_aggregated* creates groups of parallel execution agents that execute in threads implicitly created by the executor. This executor prefers continuing with the creating thread first before executing newly created threads.

This executor conforms to the concepts of a TwoWayExecutor, and a BulkTwoWayExecutor

Public Types

```
template<>
using execution_category = hpx::execution::parallel_execution_tag
    Associate the parallel_execution_tag executor tag type as a default with this executor.
```

```
template<>
using executor_parameters_type = hpx::execution::static_chunk_size
    Associate the static_chunk_size executor parameters type as a default with this executor.
```

Public Functions

```
constexpr parallel_policy_executor_aggregated(std::size_t spread =  
                                             4, std::size_t tasks =  
                                             std::size_t(-1))
```

Create a new parallel executor.

```
template<typename F, typename S, typename ...Ts>  
std::vector<hpx::future<void>> bulk_async_execute(F &&f, S const &shape,  
                                                    Ts&&... ts) const
```

```
template<>  
struct parallel_policy_executor_aggregated<hpx::launch>
```

Public Types

```
template<>  
using execution_category = hpx::execution::parallel_execution_tag  
Associate the parallel_execution_tag executor tag type as a default with this executor.
```

```
template<>  
using executor_parameters_type = hpx::execution::static_chunk_size  
Associate the static_chunk_size executor parameters type as a default with this executor.
```

Public Functions

```
constexpr parallel_policy_executor_aggregated(hpx::launch l =  
                                              hpx::launch::async_policy{},  
                                              std::size_t spread =  
                                              4, std::size_t tasks =  
                                              std::size_t(-1))
```

Create a new parallel executor.

```
template<typename F, typename S, typename ...Ts>  
std::vector<hpx::future<void>> bulk_async_execute(F &&f, S const &shape,  
                                                    Ts&&... ts) const
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

```
class restricted_thread_pool_executor
```

Public Types

typedef *hpx::execution::parallel_execution_tag execution_category*
 Associate the parallel_execution_tag executor tag type as a default with this executor.

typedef *hpx::execution::static_chunk_size executor_parameters_type*
 Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

```
restricted_thread_pool_executor(std::size_t first_thread = 0, std::size_t num_threads = 1, threads::thread_priority priority = threads::thread_priority::default_, threads::thread_stacksize stacksize = threads::thread_stacksize::default_, threads::thread_schedule_hint schedule_hint = {}, std::size_t hierarchical_threshold = hierarchical_threshold_default_)
```

Create a new parallel executor.

```
restricted_thread_pool_executor(restricted_thread_pool_executor const &other)
```

Private Members

```
threads::thread_pool_base *pool_ = nullptr  

threads::thread_priority priority_ = threads::thread_priority::default_  

threads::thread_stacksize stacksize_ = threads::thread_stacksize::default_  

threads::thread_schedule_hint schedulehint_ = {}  

std::size_t hierarchical_threshold_ = hierarchical_threshold_default_  

std::size_t first_thread_  

std::size_t num_threads_  

std::atomic<std::size_t> os_thread_
```

Private Static Attributes

```
constexpr std::size_t hierarchical_threshold_default_ = 6  

namespace hpx  
  

namespace execution  
  

namespace experimental
```

Functions

```
template<typename BaseScheduler>
scheduler_executor(BaseScheduler &&sched)
```

```
template<typename BaseScheduler>
struct scheduler_executor
```

Public Functions

```
constexpr scheduler_executor()
```

```
template<typename Scheduler, typename Enable = std::enable_if_t<hpx::execution::experimental::is_scheduler_v<Scheduler>>
constexpr scheduler_executor(Scheduler &&sched)
```

```
constexpr scheduler_executor(scheduler_executor&&)
```

```
constexpr scheduler_executor&operator=(scheduler_executor&&)
```

```
constexpr scheduler_executor(scheduler_executor const&)
```

```
constexpr scheduler_executor&operator=(scheduler_executor const&)
```

```
namespace hpx
```

```
namespace execution
```

```
struct sequenced_executor
```

```
#include <sequenced_executor.hpp> A sequential_executor creates groups of sequential execution agents which execute in the calling thread. The sequential order is given by the lexicographical order of indices in the index space.
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

TypeDefs

```
using thread_pool_executor = hpx::execution::parallel_executor
```

```
namespace hpx
```

```
namespace execution
```

```
namespace experimental
```

```
struct thread_pool_scheduler
```

Public Functions

```
constexpr thread_pool_scheduler()
thread_pool_scheduler(hpx::threads::thread_pool_base *pool)

namespace hpx

namespace execution

namespace experimental
```

Functions

```
template<typename Sender, typename Shape, typename F>
constexpr auto tag_invoke(bulk_t, thread_pool_scheduler scheduler, Sender &&sender,
                           Shape &&shape, F &&f)
```

filesystem

See *Public API* for a list of names and headers that are part of the public *HPX API*.

This file provides a compatibility layer using Boost.Filesystem for the C++17 filesystem library. It is *not* intended to be a complete compatibility layer. It only contains functions required by the HPX codebase. It also provides some functions only available in Boost.Filesystem when using C++17 filesystem.

```
namespace hpx
```

```
namespace filesystem
```

Functions

```
path initial_path()
std::string basename(path const &p)
path canonical(path const &p, path const &base)
path canonical(path const &p, path const &base, std::error_code &ec)
```

format

See *Public API* for a list of names and headers that are part of the public *HPX API*.

Defines

```
DECL_TYPE_SPECIFIER (Type, Spec)
```

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename ...Args>
```

```
std::string format (boost::string_ref format_str, Args const&... args)
```

```
template<typename ...Args>
```

```
std::ostream &format_to (std::ostream &os, boost::string_ref format_str, Args const&... args)
```

```
template<typename Range>
```

```
detail::format_join<Range> format_join (Range const &range, boost::string_ref delimiter)
```

```
namespace hpx
```

```
namespace util
```

```
class bad_lexical_cast : public bad_cast
```

Public Functions

```
bad_lexical_cast ()
```

```
const char *what () const
```

```
virtual ~bad_lexical_cast ()
```

```
bad_lexical_cast (std::type_info const &source_type_arg, std::type_info const &target_type_arg)
```

```
std::type_info const &source_type () const
```

```
std::type_info const &target_type () const
```

Private Members

```
std::type_info const *source
```

```
std::type_info const *target
```

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename T, typename Char>
T from_string(std::basic_string<Char> const &v)

template<typename T, typename U, typename Char>
T from_string(std::basic_string<Char> const &v, U &&default_value)

template<typename T>
T from_string(std::string const &v)

template<typename T, typename U>
T from_string(std::string const &v, U &&default_value)

namespace hpx
```

```
namespace util
```

Functions

```
template<typename T>
std::string to_string(T const &v)
```

functional

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Archive, typename F, typename ...Ts>
void serialize(Archive &ar, ::hpx::util::detail::bound<F, Ts...> &bound, unsigned int const
               version = 0)

template<typename Archive, std::size_t I>
void serialize(Archive&, ::hpx::util::detail::placeholder<I>&, unsigned int const = 0)
```

```
namespace util
```

Functions

```
template<typename F, typename ...Ts, typename Enable = std::enable_if_t<!traits::is_action_v<std::decay_t<F>>>
constexpr detail::bound<std::decay_t<F>, util::make_index_pack_t<sizeof...(Ts)>, util::decay_unwrap_t<Ts>...> bind
```

```
namespace placeholders
```

Variables

```
constexpr detail::placeholder<1> _1 = {}
constexpr detail::placeholder<2> _2 = {}
constexpr detail::placeholder<3> _3 = {}
constexpr detail::placeholder<4> _4 = {}
constexpr detail::placeholder<5> _5 = {}
constexpr detail::placeholder<6> _6 = {}
constexpr detail::placeholder<7> _7 = {}
constexpr detail::placeholder<8> _8 = {}
constexpr detail::placeholder<9> _9 = {}

namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Archive, typename F, typename ...Ts>
void serialize(Archive &ar, ::hpx::util::detail::bound_back<F, Ts...> &bound, unsigned int
               const version = 0)

namespace util
```

Functions

```
template<typename F, typename ...Ts>
constexpr detail::bound_back<typename std::decay<F>::type, typename util::make_index_pack<sizeof...(Ts)>::type>
```

```
template<typename F>
constexpr std::decay<F>::type bind_back(F &&f)
```

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Archive, typename F, typename ...Ts>
void serialize(Archive &ar, ::hpx::util::detail::bound_front<F, Ts...> &bound, unsigned int
               const version = 0)

namespace util
```

Functions

```
template<typename F, typename ...Ts>
constexpr detail::bound_front<typename std::decay<F>::type, typename util::make_index_pack<sizeof...(Ts)>::type>
```

```
template<typename F>
constexpr std::decay<F>::type bind_front(F &&f)

namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Archive, typename F, typename ...Ts>
void serialize(Archive &ar, ::hpx::util::detail::deferred<F, Ts...> &d, unsigned int const ver-
               sion = 0)

namespace util
```

Functions

```
template<typename F, typename ...Ts>
detail::deferred<std::decay_t<F>, util::make_index_pack_t<sizeof...(Ts)>, util::decay_unwrap_t<Ts>...> deferred_call
```

```
template<typename F>
std::decay_t<F> deferred_call(F &&f)
```

Defines

```
HPX_UTIL_REGISTER_FUNCTION_DECLARATION(Sig, F, Name)
```

```
HPX_UTIL_REGISTER_FUNCTION(Sig, F, Name)
```

```
namespace hpx
```

```
namespace util
```

Typedefs

```
template<typename Sig>
using function_nonser = function<Sig, false>

template<typename R, typename ...Ts, bool Serializable>
class function<R (Ts...) , Serializable> : public detail::basic_function<R (Ts..., true, Serializable)>
```

Public Types

```
template<>
using result_type = R
```

Public Functions

```
constexpr function (std::nullptr_t = nullptr)
```

```
function (function const&)
```

```
function (function&&)
```

```
function &operator= (function const&)
```

```
function &operator= (function&&)
```

```
template<typename F, typename FD = typename std::decay<F>::type, typename Enable1 = typename std::enable_if<FD (F && f)>
```

```
template<typename F, typename FD = typename std::decay<F>::type, typename Enable1 = typename std::enable_if<FD (F && f)>
```

Private Types

```
template<>
using base_type = detail::basic_function<R (Ts...) , true, Serializable>
```

```
namespace hpx
```

```
namespace util
```

```
template<typename R, typename ...Ts>
class function_ref<R (Ts...) >
```

Public Functions

```
template<typename F, typename FD = typename std::decay<F>::type, typename Enable = typename std::enable_if<std::is_invocable<FD, F>::value, void>::type>
function_ref(F &&f)

function_ref(function_ref const &other)

template<typename F, typename FD = typename std::decay<F>::type, typename Enable = typename std::enable_if<std::is_invocable<FD, F>::value, void>::type>
function_ref &operator=(F &&f)

function_ref &operator=(function_ref const &other)

template<typename F, typename T = typename std::remove_reference<F>::type, typename Enable = typename std::enable_if<std::is_invocable<FD, F>::value, void>::type>
void assign(F &&f)

template<typename T>
void assign(std::reference_wrapper<T> f_ref)

template<typename T>
void assign(T *f_ptr)

void swap(function_ref &f)

R operator() (Ts... vs) const

std::size_t get_function_address() const
char const *get_function_annotation() const
util::itt::string_handle get_function_annotation_itt() const
```

Protected Attributes

```
template<>
R (*vptr)(void*, Ts&&...)
void *object
```

Private Types

```
template<>
using VTable = detail::function_ref_vtable<R (Ts...) >
```

Private Static Functions

```
template<typename T>
static VTable const *get_vtable()
```

Defines

```
HPX_INVOKE_R(R, F, ...)
```

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename F, typename ...Ts>
```

```
constexpr util::invoke_result<F, Ts...>::type invoke(F &&f, Ts&&... vs)
```

Invokes the given callable object f with the content of the argument pack vs

Return The result of the callable object when it's called with the given argument types.

Note This function is similar to `std::invoke` (C++17)

Parameters

- f: Requires to be a callable object. If f is a member function pointer, the first argument in the pack will be treated as the callee (this object).
- vs: An arbitrary pack of arguments

Exceptions

- `std::exception`: like objects thrown by call to object f with the argument types vs.

```
template<typename R, typename F, typename ...Ts>
```

```
constexpr R invoke_r(F &&f, Ts&&... vs)
```

Template Parameters

- R: The result type of the function when it's called with the content of the given argument types vs.

```
namespace functional
```

```
struct invoke
```

Public Functions

```
template<typename F, typename ...Ts>
```

```
constexpr util::invoke_result<F, Ts...>::type operator()(F &&f, Ts&&... vs) const
```

```
template<typename R>
```

```
struct invoke_r
```

Public Functions

```
template<typename F, typename ...Ts>
```

```
constexpr R operator()(F &&f, Ts&&... vs) const
```

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename F, typename Tuple>
constexpr detail::invoke_fused_result<F, Tuple>::type invoke_fused(F &&f, Tuple &&t)
    Invokes the given callable object f with the content of the sequenced type t (tuples, pairs)
```

Return The result of the callable object when it's called with the content of the given sequenced type.

Note This function is similar to `std::apply` (C++17)

Parameters

- `f`: Must be a callable object. If `f` is a member function pointer, the first argument in the sequenced type will be treated as the callee (this object).
- `t`: A type whose contents are accessible through a call to `hpx::get`.

Exceptions

- `std::exception`: like objects thrown by call to object `f` with the arguments contained in the sequenceable type `t`.

```
template<typename R, typename F, typename Tuple>
constexpr R invoke_fused_r(F &&f, Tuple &&t)
```

Template Parameters

- `R`: The result type of the function when it's called with the content of the given sequenced type.

namespace hpx

namespace util

Functions

```
template<typename M, typename C>
constexpr detail::mem_fn<M C::*> mem_fn(M C::* pm)
```

```
template<typename R, typename C, typename ...Ps>
constexpr detail::mem_fn<R (C::*) (Ps...) > mem_fn
    R (C::* pm)Ps...
```

```
template<typename R, typename C, typename ...Ps>
constexpr detail::mem_fn<R (C::*) (Ps...) > const mem_fn
    R (C::* pm)Ps... const
```

namespace hpx

namespace serialization

Functions

```
template<typename Archive, typename F>
void serialize(Archive &ar, ::hpx::util::detail::one_shot_wrapper<F> &one_shot_wrapper, unsigned int const version = 0)
```

namespace util

Functions

```
template<typename F>
constexpr detail::one_shot_wrapper<typename std::decay<F>::type> one_shot (F &&f)

namespace hpx

namespace util
```

Functions

```
template<typename T>
std::enable_if<!traits::is_bind_expression<typename std::decay<T>::type>::value, detail::protected_bind<typename std::decay<T>::type>::value> protect (T
&&v)
```

Defines

```
HPX_UTIL_REGISTER_UNIQUE_FUNCTION_DECLARATION(Sig, F, Name)
```

```
HPX_UTIL_REGISTER_UNIQUE_FUNCTION(Sig, F, Name)
```

```
namespace hpx
```

```
namespace util
```

Typedefs

```
template<typename Sig>
using unique_function_nonser = unique_function<Sig, false>

template<typename R, typename ...Ts, bool Serializable>
class unique_function<R(Ts...), Serializable> : public detail::basic_function<R
Ts..., false, Serializable>
```

Public Types

```
typedef R result_type
```

Public Functions

```
constexpr unique_function(std::nullptr_t = nullptr)
unique_function(unique_function&&)
unique_function &operator=(unique_function&&)
template<typename F, typename FD = typename std::decay<F>::type, typename Enable1 = typename std::enable_if<std::is_invocable<FD, F>::value, FD> F unique_function(F &&f)
template<typename F, typename FD = typename std::decay<F>::type, typename Enable1 = typename std::enable_if<std::is_invocable<FD, F>::value, FD> F unique_function &operator=(F &&f)
```

Private Types

```
template<>
using base_type = detail::basic_function<R (Ts...) , false, Serializable>
template<typename R, typename Obj, typename ...Ts>
struct get_function_address<R (Obj::*) (Ts...) >
```

Public Static Functions

```
static std::size_t call(R (Obj::*f)) Ts...
template<typename R, typename Obj, typename ...Ts>
struct get_function_address<R (Obj::*) (Ts...) const>
```

Public Static Functions

```
static std::size_t call(R (Obj::*f)) Ts...
const
namespace hpx
```

namespace traits

```
template<typename F, typename Enable = void>
struct get_function_address
```

Public Static Functions

```
static constexpr std::size_t call(F const &f)
template<typename R, typename ...Ts>
struct get_function_address<R (*) (Ts...) >
```

Public Static Functions

```
static constexpr std::size_t call (R (*f)) Ts...
template<typename R, typename Obj, typename ...Ts>
struct get_function_address<R (Obj::*) (Ts...) const>
```

Public Static Functions

```
static std::size_t call (R (Obj::*f)) Ts...
const
template<typename R, typename Obj, typename ...Ts>
struct get_function_address<R (Obj::*) (Ts...) >
```

Public Static Functions

```
static std::size_t call (R (Obj::*f)) Ts...
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename F, typename Enable = void>
struct get_function_annotation
```

Public Static Functions

```
static constexpr char const *call (F const&)
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_action_v = is_action<T>::value
template<typename T>
constexpr bool is_bound_action_v = is_bound_action<T>::value
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_bind_expression_v = is_bind_expression<T>::value

template<typename T>
struct is_bind_expression : public std::is_bind_expression<T>
    Subclassed by hpx::traits::is_bind_expression<T const>

namespace hpx

namespace traits

template<typename T>
struct is_placeholder
    #include <i_is_placeholder.hpp> If T is a standard, Boost, or HPX placeholder (_1, _2, _3, ...) then this template is derived from std::integral_constant<int, 1>, std::integral_constant<int, 2>, std::integral_constant<int, 3>, respectively. Otherwise it is derived from std::integral_constant<int, 0>.
```

futures

See *Public API* for a list of names and headers that are part of the public *HPX API*.

Defines

```
HPX_MAKE_EXCEPTIONAL_FUTURE(T, errorcode, f, msg)
namespace hpx
```

Functions

```
template<typename R, typename U>
hpx::future<R> make_future(hpx::future<U> &&f)

template<typename R, typename U, typename Conv>
hpx::future<R> make_future(hpx::future<U> &&f, Conv &&conv)

template<typename R, typename U>
hpx::future<R> make_future(hpx::shared_future<U> f)

template<typename R, typename U, typename Conv>
hpx::future<R> make_future(hpx::shared_future<U> f, Conv &&conv)

template<typename R>
hpx::shared_future<R> make_shared_future(hpx::future<R> &&f)

template<typename R>
hpx::shared_future<R> &make_shared_future(hpx::shared_future<R> &f)

template<typename R>
hpx::shared_future<R> &&make_shared_future(hpx::shared_future<R> &&f)
```

```
template<typename R>
hpx::shared_future<R> const &make_shared_future(hpx::shared_future<R> const &f)

template<typename T, typename Allocator, typename ...Ts>
std::enable_if_t<std::is_constructible_v<T, Ts&&...> || std::is_void_v<T>, future<T>> make_ready_future_alloc(Allocator const &a, T const &a, Ts&&...ts)

template<typename T, typename ...Ts>
std::enable_if_t<std::is_constructible_v<T, Ts&&...> || std::is_void_v<T>, future<T>> make_ready_future(Ts&&...ts)

template<int DeductionGuard = 0, typename Allocator, typename T>
future<hpx::util::decay_unwrap_t<T>> make_ready_future_alloc(Allocator const &a, T const &a, T &&init)

template<int DeductionGuard = 0, typename T>
future<hpx::util::decay_unwrap_t<T>> make_ready_future(T &&init)

template<typename T>
future<T> make_exceptional_future(std::exception_ptr const &e)

template<typename T, typename E>
future<T> make_exceptional_future(E e)

template<int DeductionGuard = 0, typename T>
future<hpx::util::decay_unwrap_t<T>> make_ready_future_at(hpx::chrono::steady_time_point const &abs_time, T &&init)

template<int DeductionGuard = 0, typename T>
future<hpx::util::decay_unwrap_t<T>> make_ready_future_after(hpx::chrono::steady_duration const &rel_time, T const &rel_time, T &&init)

template<typename Allocator>
future<void> make_ready_future_alloc(Allocator const &a)

future<void> make_ready_future()

future<void> make_ready_future_at(hpx::chrono::steady_time_point const &abs_time)

future<void> make_ready_future_after(hpx::chrono::steady_duration const &rel_time)

template<typename R>
class future : public hpx::lcos::detail::future_base<future<R>, R>
```

Public Types

```
template<>
using result_type = R

template<>
using shared_state_type = typename base_type::shared_state_type
```

Public Functions

```

constexpr future()

future(future &&other)

future(future<future> &&other)

future(future<shared_future<R>> &&other)

template<typename T>
future(future<T> &&other, std::enable_if_t<std::is_void_v<R> && !traits::is_future_v<T>, T>*>
    = nullptr)

~future()

future &operator=(future &&other)

shared_future<R> share()

hpx::traits::future_traits<future>::result_type get()

hpx::traits::future_traits<future>::result_type get(error_code &ec)

template<typename F>
decltype(auto) then(F &&f, error_code &ec = throws)

template<typename T0, typename F>
decltype(auto) then(T0 &&t0, F &&f, error_code &ec = throws)

template<typename Allocator, typename F>
auto then_alloc(Allocator const &alloc, F &&f, error_code &ec = throws)

```

Private Types

```

template<>
using base_type = lcos::detail::future_base<future<R>, R>

```

Private Functions

```

future(hpx::intrusive_ptr<shared_state_type> const &state)

future(hpx::intrusive_ptr<shared_state_type> &&state)

template<typename SharedState>
future(hpx::intrusive_ptr<SharedState> const &state)

```

Friends

```
friend hpx::hpx::traits::future_access

template<typename Receiver>
lcos::detail::operation_state<Receiver, future> tag_invoke(hpx::execution::experimental::connect_t,
    future &&f, Receiver &&receiver)

struct invalidate
```

Public Functions

```
template<>
constexpr invalidate(future &f)

template<>
~invalidate()
```

Public Members

```
template<>
future &f_

template<typename R>
class shared_future : public hpx::lcos::detail::future_base<shared_future<R>, R>
```

Public Types

```
template<>
using result_type = R

template<>
using shared_state_type = typename base_type::shared_state_type
```

Public Functions

```
constexpr shared_future()

shared_future(shared_future const &other)

shared_future(shared_future &&other)

shared_future(future<R> &&other)

shared_future(future<shared_future> &&other)

template<typename T>
shared_future(shared_future<T> const &other, std::enable_if_t<std::is_void_v<R> &&
    !traits::is_future_v<T>, T>* = nullptr)

~shared_future()

shared_future &operator=(shared_future const &other)

shared_future &operator=(shared_future &&other)
```

```

get () const
get (error_code &ec) const
template<typename F>
decltype(auto) then (F &&f, error_code &ec = throws) const

template<typename T0, typename F>
decltype(auto) then (T0 &&t0, F &&f, error_code &ec = throws) const

template<typename Allocator, typename F>
auto then_alloc (Allocator const &alloc, F &&f, error_code &ec = throws)

```

Private Types

```

template<>
using base_type = lcos::detail::future_base<shared_future<R>, R>

```

Private Functions

```

shared_future (hpx::intrusive_ptr<shared_state_type> const &state)
shared_future (hpx::intrusive_ptr<shared_state_type> &&state)
template<typename SharedState>
shared_future (hpx::intrusive_ptr<SharedState> const &state)

```

Friends

```

friend hpx::hpx::traits::future_access

template<typename Receiver>
lcos::detail::operation_state<Receiver, shared_future> tag_invoke (hpx::execution::experimental::connect_t,
shared_future &&f, Receiver
&&receiver)

template<typename Receiver>
lcos::detail::operation_state<Receiver, shared_future> tag_invoke (hpx::execution::experimental::connect_t,
shared_future &f, Receiver
&&receiver)

namespace lcos

```

Functions

```

template<typename R, typename U>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_
template<typename R, typename U, typename Conv>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_
template<typename T, typename Allocator, typename... Ts>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_
template<typename T, typename... Ts>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_
template<typename T>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_exceptional_
template<typename T, typename E>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_

```

```
template<int DeductionGuard = 0, typename T>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_ready_future<T>")  
template<int DeductionGuard = 0, typename T>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_ready_future<T>")  
template<typename Allocator>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_ready_future<T, Allocator>")  
template<typename T>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_ready_future<T>")  
template<typename T>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_ready_future<T>")  
template<typename T>hpx::lcos::HPX_DEPRECATED_V(1, 8, "hpx::lcos::make_ready_future<T>")  
  
namespace serialization
```

Functions

```
template<typename Archive, typename T>  
void serialize(Archive &ar, ::hpx::future<T> &f, unsigned version)  
  
template<typename Archive, typename T>  
void serialize(Archive &ar, ::hpx::shared_future<T> &f, unsigned version)  
  
namespace hpx
```

```
namespace lcos
```

TypeDefs

```
typedef hpx::shared_future<R> instead  
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```
template<typename Result, bool Cancelable>  
class futures_factory<Result(), Cancelable>
```

Public Functions

```
futures_factory()
```

```
template<typename Executor, typename F>  
futures_factory(Executor &exec, F &&f)
```

```
template<typename Executor>  
futures_factory(Executor &exec, Result (*f))
```

```
template<typename F, typename Enable = std::enable_if_t<!std::is_same_v<std::decay_t<F>, futures_factory>>  
futures_factory(F &&f)
```

```
futures_factory(Result (*f))
```

```

~futures_factory()

futures_factory (futures_factory const &rhs)
futures_factory &operator= (futures_factory const &rhs)
futures_factory (futures_factory &&rhs)
futures_factory &operator= (futures_factory &&rhs)
void operator() () const

threads::thread_id_ref_type apply (const char *annotation = "futures_factory::apply",
                                    launch policy = launch::async, error_code &ec =
                                    throws) const

threads::thread_id_ref_type apply (threads::thread_pool_base *pool, const char *annotation =
                                    = "futures_factory::apply", launch policy =
                                    launch::async, error_code &ec = throws) const

hpx::future<Result> get_future (error_code &ec = throws)

constexpr bool valid() const

void set_exception (std::exception_ptr const &e)

```

Protected Types

```

template<>
using task_impl_type = lcos::detail::task_base<Result>

```

Protected Attributes

```

hpx::intrusive_ptr<task_impl_type> task_
bool future_obtained_ = false

```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```

template<typename R, typename ...Ts>
class packaged_task<R (Ts...)>

```

Public Functions

```
packaged_task()

template<typename F, typename FD = std::decay_t<F>, typename Enable = std::enable_if_t<!std::is_same_v<FD, F>>
packaged_task(F &&f)

template<typename Allocator, typename F, typename FD = std::decay_t<F>, typename Enable = std::enable_if_t<!std::is_same_v<FD, F>>
packaged_task(Allocator_arg_t, Allocator const &a, F &&f)

packaged_task(packaged_task const &rhs)

packaged_task(packaged_task &&rhs)

packaged_task &operator=(packaged_task const &rhs)

packaged_task &operator=(packaged_task &&rhs)

void swap(packaged_task &rhs)

void operator()(Ts... ts)

hpx::future<R> get_future(error_code &ec = throws)

bool valid() const

void reset(error_code &ec = throws)

void set_exception(std::exception_ptr const &e)
```

Private Types

```
template<>
using function_type = util::unique_function_nonser<R (Ts...) >
```

Private Members

```
function_type function_
local::promise<R> promise_
```

```
namespace std
```

Functions

```
template<typename Sig>
void swap(hpx::lcos::local::packaged_task<Sig> &lhs, hpx::lcos::local::packaged_task<Sig> &rhs)
```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

Functions

```
template<typename R>
void swap(promise<R> &x, promise<R> &y)

template<typename R>
class promise : public hpx::lcos::local::detail::promise_base<R>
```

Public Functions

```
promise()

template<typename Allocator>
promise(std::allocator_arg_t, Allocator const &a)

promise(promise &&other)

~promise()

promise &operator=(promise &&other)

void swap(promise &other)

void set_value(R const &r)

void set_value(R &&r)

template<typename ...Ts>
void set_value(Ts&&... ts)
```

Private Types

```
template<>
using base_type = detail::promise_base<R>

template<typename R>
class promise<R&> : public hpx::lcos::local::detail::promise_base<R&>
```

Public Functions

```
promise()

template<typename Allocator>
promise(std::allocator_arg_t, Allocator const &a)

promise(promise &&other)

~promise()

promise &operator=(promise &&other)

void swap(promise &other)

void set_value(R &r)
```

Private Types

```
template<>
using base_type = detail::promise_base<R&>

template<>
class promise<void> : public hpx::lcos::local::detail::promise_base<void>
```

Public Functions

```
promise()

template<typename Allocator>
promise(std::allocator_arg_t, Allocator const &a)

promise(promise &&other)

~promise()

promise &operator=(promise &&other)

void swap(promise &other)

void set_value()
```

Private Types

```
template<>
using base_type = detail::promise_base<void>

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename T>
using acquire_future_t = typename acquire_future<T>::type

struct acquire_future_disp
```

Public Functions

```
template<typename T>
acquire_future_t<T> operator() (T &&t) const

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename T>
using acquire_shared_state_t = typename acquire_shared_state<T>::type

struct acquire_shared_state_disp
```

Public Functions

```
template<typename T>
acquire_shared_state_t<T> operator()(T &&t) const

template<typename R>
struct future_access<hpx::future<R>>
```

Public Static Functions

```
template<typename SharedState>
static hpx::future<R> create(hpx::intrusive_ptr<SharedState> const &shared_state)

template<typename T = void>
static hpx::future<R> create(detail::shared_state_ptr_for_t<hpx::future<hpx::future<R>>>
    const &shared_state)

template<typename SharedState>
static hpx::future<R> create(hpx::intrusive_ptr<SharedState> &&shared_state)

template<typename T = void>
static hpx::future<R> create(detail::shared_state_ptr_for_t<hpx::future<hpx::future<R>>>
    &&shared_state)

template<typename SharedState>
static hpx::future<R> create(SharedState *shared_state, bool addref = true)

static traits::detail::shared_state_ptr_t<R> const &get_shared_state(hpx::future<R> const
    &f)

static traits::detail::shared_state_ptr_t<R>::element_type *detach_shared_state(hpx::future<R>
    &&f)

template<typename Destination>
static void transfer_result(hpx::future<R> &&src, Destination &dest)
```

Private Static Functions

```
template<typename Destination>
static void transfer_result_impl(hpx::future<R> &&src, Destination &dest, std::false_type)

template<typename Destination>
static void transfer_result_impl(hpx::future<R> &&src, Destination &dest, std::true_type)

template<typename R>
struct future_access<hpx::shared_future<R>>
```

Public Static Functions

```
template<typename SharedState>
static hpx::shared_future<R> create (hpx::intrusive_ptr<SharedState> const &shared_state)

template<typename T = void>
static hpx::shared_future<R> create (detail::shared_state_ptr_for_t<hpx::shared_future<hpx::future<R>>>
                                         const &shared_state)

template<typename SharedState>
static hpx::shared_future<R> create (hpx::intrusive_ptr<SharedState> &&shared_state)

template<typename T = void>
static hpx::shared_future<R> create (detail::shared_state_ptr_for_t<hpx::shared_future<hpx::future<R>>>
                                         &&shared_state)

template<typename SharedState>
static hpx::shared_future<R> create (SharedState *shared_state, bool addref = true)

static traits::detail::shared_state_ptr_t<R> const &get_shared_state (hpx::shared_future<R>
                                         const &f)

static traits::detail::shared_state_ptr_t<R>::element_type *detach_shared_state (hpx::shared_future<R>
                                         const &f)

template<typename Destination>
static void transfer_result (hpx::shared_future<R> &&src, Destination &dest)
```

Private Static Functions

```
template<typename Destination>
static void transfer_result_impl (hpx::shared_future<R> &&src, Destination &dest,
                                    std::false_type)

template<typename Destination>
static void transfer_result_impl (hpx::shared_future<R> &&src, Destination &dest,
                                    std::true_type)

namespace hpx

namespace traits
```

Typedefs

```
template<typename SharedState, typename Allocator>
using shared_state_allocator_t = typename shared_state_allocator<SharedState, Allocator>::type
```

Variables

```
template<typename R>
constexpr bool is_shared_state_v = is_shared_state<R>::value

template<typename R>
struct future_access<hpx::future<R>>
```

Public Static Functions

```
template<typename SharedState>
static hpx::future<R> create (hpx::intrusive_ptr<SharedState> const &shared_state)
```

```
template<typename T = void>
static hpx::future<R> create (detail::shared_state_ptr_for_t<hpx::future<hpx::future<R>>>
                                const &shared_state)
```

```
template<typename SharedState>
static hpx::future<R> create (hpx::intrusive_ptr<SharedState> &&shared_state)
```

```
template<typename T = void>
static hpx::future<R> create (detail::shared_state_ptr_for_t<hpx::future<hpx::future<R>>>
                                &&shared_state)
```

```
template<typename SharedState>
static hpx::future<R> create (SharedState *shared_state, bool addrf = true)
```

```
static traits::detail::shared_state_ptr_t<R> const &get_shared_state (hpx::future<R>
                                                                     const &f)
```

```
static traits::detail::shared_state_ptr_t<R>::element_type *detach_shared_state (hpx::future<R>
                                         &&f)
```

```
template<typename Destination>
static void transfer_result (hpx::future<R> &&src, Destination &dest)
```

Private Static Functions

```
template<typename Destination>
static void transfer_result_impl (hpx::future<R> &&src, Destination &dest,
                                    std::false_type)
```

```
template<typename Destination>
static void transfer_result_impl (hpx::future<R> &&src, Destination &dest,
                                    std::true_type)
```

```
template<typename R>
struct future_access<hpx::shared_future<R>>
```

Public Static Functions

```
template<typename SharedState>
static hpx::shared_future<R> create (hpx::intrusive_ptr<SharedState> const
                                         &shared_state)

template<typename T = void>
static hpx::shared_future<R> create (detail::shared_state_ptr_for_t<hpx::shared_future<hpx::future<R>>>
                                         const &shared_state)

template<typename SharedState>
static hpx::shared_future<R> create (hpx::intrusive_ptr<SharedState> &&shared_state)

template<typename T = void>
static hpx::shared_future<R> create (detail::shared_state_ptr_for_t<hpx::shared_future<hpx::future<R>>>
                                         &&shared_state)

template<typename SharedState>
static hpx::shared_future<R> create (SharedState *shared_state, bool addref = true)

static traits::detail::shared_state_ptr_t<R> const &get_shared_state (hpx::shared_future<R> const
                                         &f)

static traits::detail::shared_state_ptr_t<R>::element_type *detach_shared_state (hpx::shared_future<R> const
                                         &f)

template<typename Destination>
static void transfer_result (hpx::shared_future<R> &&src, Destination &dest)
```

Private Static Functions

```
template<typename Destination>
static void transfer_result_impl (hpx::shared_future<R> &&src, Destination &dest,
                                    std::false_type)

template<typename Destination>
static void transfer_result_impl (hpx::shared_future<R> &&src, Destination &dest,
                                    std::true_type)

namespace hpx
```

```
    namespace traits
```

TypeDefs

```
template<typename Future, typename F>
using future_then_result_t = typename future_then_result<Future, F>::type

template<typename R>
struct future_traits<hpx::future<R>>
```

Public Types

```
template<>
using type = R

template<>
using result_type = R

template<typename R>
struct future_traits<hpx::shared_future<R>>
```

Public Types

```
template<>
using type = R

template<>
using result_type = R const&

template<>
struct future_traits<hpx::shared_future<void>>
```

Public Types

```
template<>
using type = void

template<>
using result_type = void
```

namespace hpx

namespace traits

Typedefs

```
template<typename Future>
using future_traits_t = typename future_traits<Future>::type

template<typename R>
struct future_traits<hpx::future<R>>
```

Public Types

```
template<>
using type = R

template<>
using result_type = R

template<typename R>
struct future_traits<hpx::shared_future<R>>
```

Public Types

```
template<>
using type = R

template<>
using result_type = R const&

template<>
struct future_traits<hpx::shared_future<void>>
```

Public Types

```
template<>
using type = void

template<>
using result_type = void

namespace hpx

namespace traits

template<typename Result, typename RemoteResult, typename Enable = void>
struct get_remote_result
```

Public Static Functions

```
static Result call (RemoteResult const &rhs)

static Result call (RemoteResult &&rhs)

template<typename Result>
struct get_remote_result<Result, Result>
```

Public Static Functions

```
static Result const &call (Result const &rhs)

static Result &&call (Result &&rhs)

namespace hpx

namespace traits
```

Variables

```
template<typename R>
constexpr bool is_future_v = is_future<R>::value

template<typename R>
constexpr bool is_ref_wrapped_future_v = is_ref_wrapped_future<R>::value

template<typename Future>
struct is_future : public hpx::traits::detail::is_future_customization_point<Future>
    Subclassed by hpx::traits::is_ref_wrapped_future< std::reference_wrapper< Future > >

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename R>
using is_future_range_t = typename is_future_range<R>::type

template<typename R>
using is_ref_wrapped_future_range_t = typename is_ref_wrapped_future_range<R>::type
```

Variables

```
template<typename R>
constexpr bool is_future_range_v = is_future_range<R>::value

template<typename R>
constexpr bool is_ref_wrapped_future_range_v = is_ref_wrapped_future_range<R>::value

template<typename R>
struct future_range_traits<R, true>
```

Public Types

```
template<>
using future_type = typename range_traits<R>::value_type

template<>
struct promise_local_result<util::unused_type>
```

Public Types

```
template<>
using type = void
```

```
namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename Result>
using promise_local_result_t = typename promise_local_result<Result>::type

template<typename Result, typename Enable = void>
struct promise_local_result
```

Public Types

```
template<>
using type = Result

template<>
struct promise_local_result<util::unused_type>
```

Public Types

```
template<>
using type = void

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename Result>
using promise_remote_result_t = typename promise_remote_result<Result>::type

template<typename Result, typename Enable = void>
struct promise_remote_result
```

Public Types

```
typedef Result type

template<>
struct promise_remote_result<void>
```

Public Types

```
typedef hpx::util::unused_type type
```

hardware

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
    namespace util
```

```
        namespace hardware
```

Functions

```
template<typename T, typename U>
bool has_bit_set (T value, U bit)
```

```
template<std::size_t N, typename T>
T unbounded_shl (T x)
```

```
template<std::size_t N, typename T>
T unbounded_shr (T x)
```

```
template<std::size_t Low, std::size_t High, typename Result, typename T>
Result get_bit_range (T x)
```

```
template<std::size_t Low, typename Result, typename T>
Result pack_bits (T x)
```

```
template<std::size_t N, typename T>
struct unbounded_shifter
```

Public Static Functions

```
    static T shl (T x)
```

```
    static T shr (T x)
```

```
template<typename T>
struct unbounded_shifter<0, T>
```

Public Static Functions

```
    static T shl (T x)
```

```
    static T shr (T x)
```

```
namespace hpx
```

```
    namespace util
```

```
        namespace hardware
```

Functions

```
void cpuid(std::uint32_t (&cpuinfo)[4], std::uint32_t eax)  
void cpuidex(std::uint32_t (&cpuinfo)[4], std::uint32_t eax, std::uint32_t ecx)  
struct cpuid_register
```

Public Types

enum info

Values:

```
eax = 0  
ebx = 1  
ecx = 2  
edx = 3
```

```
namespace hpx
```

```
namespace util
```

```
namespace hardware
```

Functions

```
HPX_DEVICE std::uint64_t hpx::util::hardware::timestamp_cuda()  
namespace hpx
```

```
namespace util
```

```
namespace hardware
```

Functions

```
std::uint64_t timestamp()
```

hashing

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

Functions

```
template<std::uint64_t N>
constexpr std::uint64_t fibhash (std::uint64_t i)

namespace hpx

namespace util

class jenkins_hash
    #include <jenkins_hash.hpp> The jenkins_hash class encapsulates a hash calculation function published by Bob Jenkins here: http://burtleburtle.net/bob/hash
```

Public Types

enum seedenum

The seedenum is used as a dummy parameter to distinguish the different constructors

Values:

seed = 1

typedef std::uint32_t size_type

this is the type representing the result of this hash

Public Functions

jenkins_hash ()

constructors and destructor

jenkins_hash (size_type size)

jenkins_hash (size_type seedval, seedenum)

~jenkins_hash ()

size_type operator () (std::string const &key) const

calculate the hash value for the given key

size_type operator () (char const *key) const

bool reset (size_type size)

re-seed the hash generator

void set_seed (size_type seedval)

initialize the hash generator to a specific seed

void swap (jenkins_hash &rhs)

support for std::swap

Protected Functions

```
size_type hash (const char *k, std::size_t length) const
```

Private Members

```
size_type seed_
```

include_local

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
namespace hpx
```

TypeDefs

```
template<typename OnCompletion = lcos::local::detail::empty_oncompletion>
using barrier = lcos::local::cpp20_barrier<OnCompletion>
```

```
namespace hpx
```

TypeDefs

```
using latch = hpx::lcos::local::cpp20_latch
```

```
namespace hpx
```

TypeDefs

```
template<std::ptrdiff_t Least.MaxValue = PTRDIFF_MAX>
using counting_semaphore = hpx::lcos::local::cpp20_counting_semaphore<Least.MaxValue>
using binary_semaphore = hpx::lcos::local::cpp20_binary_semaphore<>
```

```
namespace hpx
```

TypeDefs

```
using task_cancelled_exception = hpx::parallel::task_canceled_exception
```

ini

See *Public API* for a list of names and headers that are part of the public *HPX API*.

Defines

```
HPX_SECTION_VERSION
```

```
namespace hpx
```

```
    namespace util
```

```
        class section
```

Public Types

```
typedef util::function_nonser<void (std::string const&, std::string const&) >
    entry_changed_func
typedef std::pair<std::string, entry_changed_func> entry_type
typedef std::map<std::string, entry_type> entry_map
typedef std::map<std::string, section> section_map
```

Public Functions

```
section()
section (std::string const &filename, section *root = nullptr)
section (section const &in)
~section()
section &operator= (section const &rhs)

void parse (std::string const &sourcename, std::vector<std::string> const &lines, bool verify_existing = true, bool weed_out_comments = true, bool replace_existing = true)
void parse (std::string const &sourcename, std::string const &line, bool verify_existing = true, bool weed_out_comments = true, bool replace_existing = true)

void read (std::string const &filename)
void merge (std::string const &second)
void merge (section &second)
void dump (int ind = 0) const
void dump (int ind, std::ostream &strm) const
void add_section (std::string const &sec_name, section &sec, section *root = nullptr)
section *add_section_if_new (std::string const &sec_name)
bool has_section (std::string const &sec_name) const
section *get_section (std::string const &sec_name)
section const *get_section (std::string const &sec_name) const
```

```
section_map &get_sections()  
section_map const &get_sections() const  
void add_entry(std::string const &key, entry_type const &val)  
void add_entry(std::string const &key, std::string const &val)  
bool has_entry(std::string const &key) const  
std::string get_entry(std::string const &key) const  
std::string get_entry(std::string const &key, std::string const &dflt) const  
template<typename T>  
std::string get_entry(std::string const &key, T dflt) const  
void add_notification_callback(std::string const &key, entry_changed_func const &callback)  
entry_map const &get_entries() const  
std::string expand(std::string const &str) const  
void expand(std::string &str, std::string::size_type len) const  
void set_root(section *r, bool recursive = false)  
section *get_root() const  
std::string get_name() const  
std::string get_parent_name() const  
std::string get_full_name() const  
void set_name(std::string const &name)
```

Protected Functions

```
void line_msg(std::string msg, std::string const &file, int lnum = 0, std::string const &line  
= "")  
section &clone_from(section const &rhs, section *root = nullptr)
```

Private Types

```
using mutex_type = util::spinlock
```

Private Functions

```

section *this_()

template<typename Archive>
void save (Archive &ar, const unsigned int version) const

template<typename Archive>
void load (Archive &ar, const unsigned int version)

void add_section (std::unique_lock<mutex_type> &l, std::string const &sec_name, section &sec, section *root = nullptr)
bool has_section (std::unique_lock<mutex_type> &l, std::string const &sec_name) const
section *get_section (std::unique_lock<mutex_type> &l, std::string const &sec_name)
section const *get_section (std::unique_lock<mutex_type> &l, std::string const &sec_name) const
section *add_section_if_new (std::unique_lock<mutex_type> &l, std::string const &sec_name)
void add_entry (std::unique_lock<mutex_type> &l, std::string const &fullkey, std::string const &key, std::string val)
void add_entry (std::unique_lock<mutex_type> &l, std::string const &fullkey, std::string const &key, entry_type const &val)
bool has_entry (std::unique_lock<mutex_type> &l, std::string const &key) const
std::string get_entry (std::unique_lock<mutex_type> &l, std::string const &key) const
std::string get_entry (std::unique_lock<mutex_type> &l, std::string const &key, std::string const &dflt) const
void add_notification_callback (std::unique_lock<mutex_type> &l, std::string const &key, entry_changed_func const &callback)
std::string expand (std::unique_lock<mutex_type> &l, std::string in) const
void expand (std::unique_lock<mutex_type> &l, std::string&, std::string::size_type) const
void expand_bracket (std::unique_lock<mutex_type> &l, std::string&, std::string::size_type) const
void expand_brace (std::unique_lock<mutex_type> &l, std::string&, std::string::size_type) const
std::string expand_only (std::unique_lock<mutex_type> &l, std::string in, std::string const &expand_this) const
void expand_only (std::unique_lock<mutex_type> &l, std::string&, std::string::size_type, std::string const &expand_this) const
void expand_bracket_only (std::unique_lock<mutex_type> &l, std::string&, std::string::size_type, std::string const &expand_this) const
void expand_brace_only (std::unique_lock<mutex_type> &l, std::string&, std::string::size_type, std::string const &expand_this) const

```

Private Members

```
section *root_
entry_map entries_
section_map sections_
std::string name_
std::string parent_name_
mutex_type mtx_
```

Friends

```
friend hpx::util::hpx::serialization::access
```

init_runtime_local

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_PREFIX
HPX_APPLICATION_STRING
namespace hpx

namespace local
```

Functions

```
int init (std::function<int> hpx::program_options::variables_map&
          >f, int argc, char **argv, init_params const &params = init_params())

int init (std::function<int> int, char***
          >f, int argc, char **argv, init_params const &params = init_params())

int init (std::function<int>
          >f int argc, char **argv, init_params const &params = init_params())

int init (std::nullptr_t, int argc, char **argv, init_params const &params = init_params())

bool start (std::function<int> hpx::program_options::variables_map&
            >f, int argc, char **argv, init_params const &params = init_params())

bool start (std::function<int> int, char***
            >f, int argc, char **argv, init_params const &params = init_params())

bool start (std::function<int>
            >f int argc, char **argv, init_params const &params = init_params())

bool start (std::function<int>, int
            >f int argc, char **argv, init_params const &params = init_params())

bool start (std::nullptr_t, int argc, char **argv, init_params const &params = init_params())
```

```

int finalize(error_code &ec = throws)

int stop(error_code &ec = throws)

int suspend(error_code &ec = throws)

int resume(error_code &ec = throws)

struct init_params
```

Public Members

```

std::reference_wrapper<hpx::program_options::options_description const> desc_cmdline = detail::default_desc
std::vector<std::string> cfg
startup_function_type startup
shutdown_function_type shutdown
hpx::resource::partitioner_mode rp_mode = ::hpx::resource::mode_default
hpx::local::detail::rp_callback_type rp_callback
```

io_service

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

```
class io_service_pool
#include <io_service_pool.hpp> A pool of io_service objects.
```

Public Functions

```
HPX_NON_COPYABLE (io_service_pool)
```

```
io_service_pool(std::size_t pool_size = 2, threads::policies::callback_notifier const
&notifier = threads::policies::callback_notifier(), char const
*pool_name = "", char const *name_postfix = "")
Construct the io_service pool.
```

Parameters

- **pool_size**: [in] The number of threads to run to serve incoming requests
- **start_thread**: [in]

```
io_service_pool(threads::policies::callback_notifier const &notifier, char const
*pool_name = "", char const *name_postfix = "")
Construct the io_service pool.
```

Parameters

- **start_thread**: [in]

```
~io_service_pool()
bool run (bool join_threads = true, barrier *startup = nullptr)
    Run all io_service objects in the pool. If join_threads is true this will also wait for all threads to
    complete

bool run (std::size_t num_threads, bool join_threads = true, barrier *startup = nullptr)
    Run all io_service objects in the pool. If join_threads is true this will also wait for all threads to
    complete

void stop ()
    Stop all io_service objects in the pool.

void join ()
    Join all io_service threads in the pool.

void clear ()
    Clear all internal data structures.

void wait ()
    Wait for all work to be done.

bool stopped ()
asio::io_context &get_io_service (int index = -1)
    Get an io_service to use.

std::thread &get_os_thread_handle (std::size_t thread_num)
    access underlying thread handle

std::size_t size () const
    Get number of threads associated with this I/O service.

void thread_run (std::size_t index, barrier *startup = nullptr)
    Activate the thread index for this thread pool.

char const *get_name () const
    Return name of this pool.

void init (std::size_t pool_size)
```

Protected Functions

```
bool run_locked (std::size_t num_threads, bool join_threads, barrier *startup)
void stop_locked ()
void join_locked ()
void clear_locked ()
void wait_locked ()
```

Private Types

```
using io_service_ptr = std::unique_ptr<asio::io_context>
using work_type = asio::io_context::work
```

Private Functions

```
work_type initialize_work (asio::io_context &io_service)
```

Private Members

std::mutex mtx_
std::vector<io_service_ptr> io_services_
 The pool of io_services.

std::vector<std::thread> threads_

std::vector<work_type> work_
 The work that keeps the io_services running.

std::size_t next_io_service_
 The next io_service to use for a connection.

bool stopped_
 set to true if stopped

std::size_t pool_size_
 initial number of OS threads to execute in this pool

threads::policies::callback_notifier const ¬ifier_
 call this for each thread start/stop

*char const *pool_name_*

*char const *pool_name_postfix_*

bool waiting_
 Set to true if waiting for work to finish.

std::unique_ptr<barrier> wait_barrier_

std::unique_ptr<barrier> continue_barrier_

iterator_support

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_ITERATOR_TRAVERSAL_TAG_NS

namespace hpx

    namespace iterators

        struct bidirectional_traversal_tag : public hpx::iterators::forward_traversal_tag
            Subclassed by hpx::iterators::random_access_traversal_tag

        struct forward_traversal_tag : public hpx::iterators::single_pass_traversal_tag
            Subclassed by hpx::iterators::bidirectional_traversal_tag

        struct incrementable_traversal_tag : public hpx::iterators::no_traversal_tag
            Subclassed by hpx::iterators::single_pass_traversal_tag

        struct no_traversal_tag
            Subclassed by hpx::iterators::incrementable_traversal_tag

        struct single_pass_traversal_tag : public hpx::iterators::incrementable_traversal_tag
            Subclassed by hpx::iterators::forward_traversal_tag

namespace traits
```

TypeDefs

```
template<typename Traversal>
using pure_traversal_tag = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::pure_traversal_tag<Traversal>

template<typename Traversal>
using pure_traversal_t = typename pure_traversal_tag<Traversal>::type

template<typename Iterator>
using pure_iterator_traversal = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::pure_iterator_traversal<Iterator>

template<typename Iterator>
using pure_iterator_traversal_t = typename pure_iterator_traversal<Iterator>::type

template<typename Cat>
using iterator_category_to_traversal = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::iterator_category_to_traversal<Cat>

template<typename Cat>
using iterator_category_to_traversal_t = typename iterator_category_to_traversal<Cat>::type

template<typename Iterator>
using iterator_traversal = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::iterator_traversal<Iterator>

template<typename Iterator>
using iterator_traversal_t = typename iterator_traversal<Iterator>::type
```

Variables

```
template<typename Traversal>
constexpr bool pure_traversal_tag_v = pure_traversal_tag<Traversal>::value

template<typename Iterator>
constexpr bool pure_iterator_traversal_v = pure_iterator_traversal<Iterator>::value

template<typename Cat>
constexpr bool iterator_category_to_traversal_v = iterator_category_to_traversal<Cat>::value

template<typename Iterator>
constexpr bool iterator_traversal_v = iterator_traversal<Iterator>::value

template<typename Incrementable, typename CategoryOrTraversal, typename Difference>
class counting_iterator<Incrementable, CategoryOrTraversal, Difference, typename std::enable_if<std::is_integral<Incrementable> && std::is_arithmetic<CategoryOrTraversal> && std::is_arithmetic<Difference>>::type>
```

Public Functions

```
counting_iterator()
counting_iterator(counting_iterator &&rhs)
counting_iterator &operator=(counting_iterator &&rhs)
counting_iterator(counting_iterator const &rhs)
counting_iterator &operator=(counting_iterator const &rhs)
counting_iterator(Incrementable x)
```

Private Types

```
template<>
using base_type = typename detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>::type
```

Private Functions

```
template<typename Iterator>
bool equal (Iterator const &rhs) const

void increment ()

void decrement ()

template<typename Distance>
void advance (Distance n)

base_type::reference dereference () const

template<typename OtherIncrementable>
base_type::difference_type distance_to (counting_iterator<OtherIncrementable, CategoryOrTraversal, Difference> const &y) const
```

Friends

```
friend iterator_core_access  
namespace hpx  
  
namespace util
```

Functions

```
template<typename Incrementable>  
counting_iterator<Incrementable> make_counting_iterator(Incrementable x)
```

```
template<typename Incrementable, typename CategoryOrTraversal, typename Difference, typename Enab  
class counting_iterator
```

Public Functions

```
counting_iterator()  
counting_iterator(counting_iterator &&rhs)  
counting_iterator &operator=(counting_iterator &&rhs)  
counting_iterator(counting_iterator const &rhs)  
counting_iterator &operator=(counting_iterator const &rhs)  
counting_iterator(Incrementable x)
```

Private Types

```
template<>  
using base_type = typename detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>
```

Private Functions

```
base_type::reference dereference() const
```

Friends

```
friend hpx::util::iterator_core_access  
template<typename Incrementable, typename CategoryOrTraversal, typename Difference>  
class counting_iterator<Incrementable, CategoryOrTraversal, Difference, typename std::enable_if<std::is_int
```

Public Functions

```
counting_iterator()
counting_iterator(counting_iterator &&rhs)
counting_iterator &operator=(counting_iterator &&rhs)
counting_iterator(counting_iterator const &rhs)
counting_iterator &operator=(counting_iterator const &rhs)
counting_iterator(Incrementable x)
```

Private Types

```
template<>
using base_type = typename detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>
```

Private Functions

```
template<typename Iterator>
bool equal(Iterator const &rhs) const
void increment()
void decrement()

template<typename Distance>
void advance(Distance n)
base_type::reference dereference() const
template<typename OtherIncrementable>
base_type::difference_type distance_to(counting_iterator<OtherIncrementable, CategoryOrTraversal, Difference> const &y) const
```

Friends

```
friend hpx::util::iterator_core_access
namespace hpx

namespace util
```

Functions

```
template<typename Generator>
generator_iterator<Generator> make_generator_iterator(Generator &gen)

template<typename Generator>
generator_iterator(Generator*)

template<typename Generator>
class generator_iterator : public hpx::util::iterator_facade<generator_iterator<Generator>, Generator::result_type>
```

Public Functions

```
generator_iterator()
generator_iterator(Generator *g)
void increment()
Generator::result_type const &dereference() const
bool equal(generator_iterator const &y) const
```

Private Types

```
template<>
using base_type = iterator_facade<generator_iterator<Generator>, typename Generator::result_type, std::forward_iterator_tag>;
```

Private Members

```
Generator *m_g
Generator::result_type m_value
namespace hpx

namespace util

template<typename Derived, typename Base, typename Value = void, typename Category = void, typename Reference = void>
class iterator_adaptor : public hpx::util::iterator_facade<Derived, value_type, iterator_category, reference_type>
    Subclassed by hpx::util::counting_iterator< Incrementable, CategoryOrTraversal, Difference, type_name std::enable_if< std::is_integral< Incrementable >::value >::type >
```

Public Types

```
template<>
using base_type = Base
```

Public Functions

```
iterator_adaptor()
iterator_adaptor(Base const &iter)
Base const &base() const
```

Protected Types

```
template<>
using base_adaptor_type = typename hpx::util::detail::iterator_adaptor_base<Derived, Base, Value, Category>;
typedef iterator_adaptor<Derived, Base, Value, Category, Reference, Difference, Pointer> iterator_adaptor
```

Protected Functions

```
Base const &base_reference() const
Base &base_reference()
```

Private Functions

```
base_adaptor_type::reference dereference() const
template<typename OtherDerived, typename OtherIterator, typename V, typename C, typename R, typename Pequal(iterator_adaptor<OtherDerived, OtherIterator, V, C, R, P> const &x) const
template<typename DifferenceType>
void advance(DifferenceType n)
void increment()
template<typename Iterator = Base, typename Enable = typename std::enable_if<traits::is_bidirectional_iterator<Iterator>>::value>
void decrement()

template<typename OtherDerived, typename OtherIterator, typename V, typename C, typename R, typename Pdistance_to(iterator_adaptor<OtherDerived, OtherIterator, V, C, R, P> const &y) const
```

Private Members

Base `iterator_`

Friends

```
friend hpx::util::hpx::util::iterator_core_access
```

Defines

```
HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD (prefix, op, result_type)
```

```
HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD_EX (prefix, op, result_type, cond)
```

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename Derived, typename T, typename Category, typename Reference, typename Distance, typename util::detail::postfix_increment_result_t<Derived, typename Derived::value_type, typename Derived::reference> operator<<
```

```
hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline, bool)
hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline, !, bool)
hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD_EX(inline)
hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD_EX(inline, bool, detail::enable_ran)
hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD_EX(inline, <=, bool, detail::enable_ran)
hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD_EX(inline, >=, bool, detail::enable_ran)
hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD_EX(inline, -, typename std::iterator<
template<typename Derived, typename T, typename Category, typename Reference, typename Distance, typename
```

```
std::enable_if_t<std::is_same_v<typename Derived::iterator_category, std::random_access_iterator_tag>, Derived> open
```

```
template<typename Derived, typename T, typename Category, typename Reference, typename Distance, typename  
std::enable_if_t<std::is_same_v<typename Derived::iterator_category, std::random_access_iterator_tag>, Derived> open
```

```
class iterator_core_access
```

Public Static Functions

```
template<typename Iterator1, typename Iterator2>  
static bool equal (Iterator1 const &lhs, Iterator2 const &rhs)  
  
template<typename Iterator>  
static void increment (Iterator &it)  
  
template<typename Iterator>  
static void decrement (Iterator &it)  
  
template<typename Reference, typename Iterator>  
static Reference dereference (Iterator const &it)
```

```
template<typename Iterator, typename Distancestatic void advance (Iterator &it, Distance n)
```

```
template<typename Iterator1, typename Iterator2static std::iterator_traits<Iterator1>::difference_type distance_to (Iterator1 const
&lhs, Iterator2 const &rhs)
```

```
template<typename Derived, typename T, typename Category, typename Reference = T&, typename Distance =
struct iterator_facade : public hpx::util::detail::iterator_facade_base<Derived, T, Category, T&, std::ptrdiff_t>
```

Subclassed by *hpx::util::iterator_adaptor< Derived, Base, Value, Category, Reference, Difference, Pointer >*

Public Types

```
template<>
using iterator_category = typename base_type::iterator_category
```

```
template<>
using value_type = typename base_type::value_type
```

```
template<>
using difference_type = typename base_type::difference_type
```

```
template<>
using pointer = typename base_type::pointer
```

```
template<>
using reference = typename base_type::reference
```

Public Functions

```
iterator_facade ()
```

Protected Types

```
template<>
using iterator_adaptor_ = iterator_facade<Derived, T, Category, Reference, Distance, Pointer>
```

Private Types

```
template<>
using base_type = detail::iterator_facade_base<Derived, T, Category, Reference, Distance, Pointer>
```

```
namespace hpx
```

```
namespace ranges
```

Typedefs

```
template<typename I, typename S = I>
using subrange_t = hpx::util::iterator_range<I, S>

namespace util
```

Functions

```
template<typename Range, typename Iterator = typename traits::range_iterator<Range>::type, typename Sentinel = typename traits::range_sentinel<Range>::type, std::enable_if<traits::is_range<Range>::value, iterator_range<Iterator, Sentinel>>::type make_iterator_range (Range &r)

template<typename Range, typename Iterator = typename traits::range_iterator<Range const>::type, typename Sentinel = typename traits::range_sentinel<Range const>::type, std::enable_if<traits::is_range<Range>::value, iterator_range<Iterator, Sentinel>>::type make_iterator_range (Range const &r)

template<typename Iterator, typename Sentinel = Iterator>
std::enable_if<traits::is_iterator<Iterator>::value, iterator_range<Iterator, Sentinel>>::type make_iterator_range (Iterator begin, Iterator end, Sentinel sentinel)

template<typename Iterator, typename Sentinel = Iterator>
class iterator_range
```

Public Functions

```
iterator_range()

iterator_range (Iterator iterator, Sentinel sentinel)

Iterator begin () const

Iterator end () const

std::ptrdiff_t size () const

bool empty () const
```

Private Members

```
Iterator _iterator
Sentinel _sentinel

namespace hpx
```

```
namespace util
```

```
namespace range_adl
```

Functions

```
template<typename C, typename Iterator = typename detail::iterator<C>::type>
constexpr Iterator begin (C &c)
```

```
template<typename C, typename Iterator = typename detail::iterator<C const>::type>
constexpr Iterator begin (C const &c)
```

```
template<typename C, typename Sentinel = typename detail::sentinel<C>::type>
constexpr Sentinel end (C &c)
```

```
template<typename C, typename Sentinel = typename detail::sentinel<C const>::type>
constexpr Sentinel end (C const &c)
```

```
template<typename C, typename Iterator = typename detail::iterator<C const>::type, typename Sentinel =
constexpr std::size_t size (C const &c)
```

```
template<typename C, typename Iterator = typename detail::iterator<C const>::type, typename Sentinel =
constexpr bool empty (C const &c)
```

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename Transformer, typename Iterator>
transform_iterator<Iterator, Transformer> make_transform_iterator (Iterator const &it,
                                                               Transformer const
                                                               &f)
```

```
template<typename Transformer, typename Iterator>
transform_iterator<Iterator, Transformer> make_transform_iterator (Iterator const &it)
```

```
template<typename Iterator, typename Transformer, typename Reference, typename Value, typename Category>
class transform_iterator
```

Public Functions

```
transform_iterator()
transform_iterator(Iterator const &it)
transform_iterator(Iterator const &it, Transformer const &f)

template<typename OtherIterator, typename OtherTransformer, typename OtherReference, typename OtherValue, typename OtherCategory, typename OtherDifference> const &t,
transform_iterator(transform_iterator<OtherIterator, OtherTransformer, OtherReference, OtherValue, OtherCategory, OtherDifference> const &t,
typename std::enable_if<std::is_convertible<OtherIterator, Iterator>::value && std::is_convertible<OtherTransformer, Transformer>::value && std::is_convertible<OtherCategory, Category>::value && std::is_convertible<OtherDifference, Difference>::value>::type* = nullptr)
```

Transformer **const &transformer() const**

Private Types

```
typedef detail::transform_iterator_base<Iterator, Transformer, Reference, Value, Category, Difference>::type base
```

Private Functions

```
base_type::reference dereference() const
```

Private Members

```
Transformer transformer_
```

Friends

```
friend hpx::util::hpx::util::iterator_core_access
```

```
template<typename ...Ts>
class zip_iterator<hpx::tuple<Ts...>> : public hpx::util::detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<hpx::tuple<
```

Public Functions

```
zip_iterator()
zip_iterator(Ts const&... vs)
zip_iterator(hpx::tuple<Ts...> &&t)
zip_iterator(zip_iterator const &other)
zip_iterator(zip_iterator &&other)
zip_iterator &operator=(zip_iterator const &other)
zip_iterator &operator=(zip_iterator &&other)
```

```
template<typename ...Ts_>
std::enable_if<std::is assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iterator_tuple>
```

```
template<typename ...Ts_>
std::enable_if<std::is assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iterator_tuple>
```

Private Types

```
template<>
using base_type = detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<hpx::tuple<Ts...>>>
```

```
template<typename F, typename ...Ts>
struct lift_zipped_iterators<F, util::zip_iterator<Ts...>>
```

Public Types

```
typedef util::zip_iterator<Ts...>::iterator_tuple_type tuple_type
```

```
typedef hpx::tuple<typename element_result_of<typename F::template apply<Ts>, Ts>::type...> result_type
```

Public Static Functions

```
template<std::size_t... Is, typename ...Ts_>
static result_type call(util::index_pack<Is...>, hpx::tuple<Ts_...> const &t)
```

```
template<typename ...Ts_>
static result_type call(util::zip_iterator<Ts_...> const &iter)
```

```
namespace hpx
```

```
namespace traits
```

```
namespace functional
```

```
template<typename F, typename ...Ts>
struct lift_zipped_iterators<F, util::zip_iterator<Ts...>>
```

Public Types

```
typedef util::zip_iterator<Ts...>::iterator_tuple_type tuple_type
```

```
typedef hpx::tuple<typename element_result_of<typename F::template apply<Ts>, Ts>::type...> result_type
```

Public Static Functions

```
template<std::size_t... Is, typename ...Ts_>
static result_type call (util::index_pack<Is...>, hpx::tuple<Ts_...> const &t)

template<typename ...Ts_>
static result_type call (util::zip_iterator<Ts_...> const &iter)

namespace util
```

Functions

```
template<typename ...Ts>
zip_iterator<typename std::decay<Ts>::type...> make_zip_iterator (Ts&&... vs)

template<typename ...Ts>
class zip_iterator : public hpx::util::detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<Ts...>>
```

Public Functions

```
zip_iterator()

zip_iterator (Ts const&... vs)

zip_iterator (hpx::tuple<Ts...> &&t)

zip_iterator (zip_iterator const &other)

zip_iterator (zip_iterator &&other)

zip_iterator &operator= (zip_iterator const &other)

zip_iterator &operator= (zip_iterator &&other)

template<typename ...Ts_>
std::enable_if<std::is assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iterator>(true) >::iterator
```

```
template<typename ...Ts_>
```

```
std::enable_if<std::is assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iterator>(true) >::iterator
```

Private Types

```
typedef detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<Ts...>> base_type

template<typename ...Ts>
class zip_iterator<hpx::tuple<Ts...>> : public hpx::util::detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<Ts...>>
```

Public Functions

```
zip_iterator()
zip_iterator(Ts const&... vs)
zip_iterator(hpx::tuple<Ts...> &&t)
zip_iterator(zip_iterator const &other)
zip_iterator(zip_iterator &&other)
zip_iterator &operator=(zip_iterator const &other)
zip_iterator &operator=(zip_iterator &&other)

template<typename ...Ts_>
std::enable_if<std::is assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iter
```

```
template<typename ...Ts_>
std::enable_if<std::is assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iter
```

Private Types

```
template<>
using base_type = detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<hpx::tuple<Ts...>>>

namespace hpx

namespace traits
```

TypeDefs

```
template<typename Iter>
using is_iterator_t = typename is_iterator<Iter>::type

template<typename Iter>
using is_output_iterator_t = typename is_output_iterator<Iter>::type

template<typename Iter>
using is_input_iterator_t = typename is_input_iterator<Iter>::type

template<typename Iter>
using is_forward_iterator_t = typename is_forward_iterator<Iter>::type

template<typename Iter>
using is_bidirectional_iterator_t = typename is_bidirectional_iterator<Iter>::type

template<typename Iter>
using is_random_access_iterator_t = typename is_random_access_iterator<Iter>::type

template<typename Iter>
using is_segmented_iterator_t = typename is_segmented_iterator<Iter>::type

template<typename Iter>
```

```

using is_segmented_local_iterator_t = typename is_segmented_local_iterator<Iter>::type
template<typename Iter>
using is_zip_iterator_t = typename is_zip_iterator<Iter>::type
template<typename Iter>
using is_contiguous_iterator_t = typename is_contiguous_iterator<Iter>::type
template<typename Iter>
using iter_value_t = typename std::iterator_traits<Iter>::value_type
template<typename Iter>
using iter_ref_t = typename std::iterator_traits<Iter>::reference

```

Variables

```

template<typename Iter>
constexpr bool is_iterator_v = is_iterator<Iter>::value
template<typename Iter, typename Category>
constexpr bool has_category_v = has_category<Iter, Category>::value
template<typename Iter>
constexpr bool is_output_iterator_v = is_output_iterator<Iter>::value
template<typename Iter>
constexpr bool is_input_iterator_v = is_input_iterator<Iter>::value
template<typename Iter>
constexpr bool is_forward_iterator_v = is_forward_iterator<Iter>::value
template<typename Iter>
constexpr bool is_bidirectional_iterator_v = is_bidirectional_iterator<Iter>::value
template<typename Iter>
constexpr bool is_random_access_iterator_v = is_random_access_iterator<Iter>::value
template<typename Iter>
constexpr bool is_segmented_iterator_v = is_segmented_iterator<Iter>::value
template<typename Iter>
constexpr bool is_segmented_local_iterator_v = is_segmented_local_iterator<Iter>::value
template<typename Iter>
constexpr bool is_zip_iterator_v = is_zip_iterator<Iter>::value
template<typename Iter>
constexpr bool is_contiguous_iterator_v = is_contiguous_iterator<Iter>::value

namespace hpx

namespace traits

```

Typedefs

```
template<typename T>
using range_iterator_t = typename range_iterator<T>::type

template<typename T>
using range_sentinel_t = typename range_sentinel<T>::type
```

Variables

```
template<typename T>
constexpr bool is_range_v = is_range<T>::value

template<typename R>
struct range_traits<R, true> : public std::iterator_traits<util::detail::iterator<R>::type>
```

Public Types

```
template<>
using iterator_type = typename util::detail::iterator<R>::type

template<>
using sentinel_type = typename util::detail::sentinel<R>::type

namespace hpx

namespace traits
```

Variables

```
template<typename Sent, typename Iter>
constexpr bool is_sentinel_for_v = is_sentinel_for<Sent, Iter>::value

template<typename Sent, typename Iter>
constexpr bool disable_sized_sentinel_for = false

template<typename Sent, typename Iter>
constexpr bool is_sized_sentinel_for_v = is_sized_sentinel_for<Sent, Iter>::value
```

Functions

```
template<typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool operator==(Iter it, sentinel<ValueType> s)

template<typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool operator==(sentinel<ValueType> s, Iter it)

template<typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool operator!=(Iter it, sentinel<ValueType> s)

template<typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool operator!=(sentinel<ValueType> s, Iter it)

template<typename ValueType>
struct sentinel
```

Public Functions

```
sentinel (ValueType stop_value)
ValueType get_stop () const
```

Private Members

```
ValueType stop
```

```
template<typename Value>
struct iterator
```

Public Types

```
template<>
using difference_type = std::ptrdiff_t
template<>
using value_type = Value
template<>
using iterator_category = std::forward_iterator_tag
template<>
using pointer = Value const*
template<>
using reference = Value const&
```

Public Functions

```
iterator (Value initialState)
virtual Value operator* () const
virtual Value operator-> () const
iterator &operator++ ()
iterator operator++ (int)
iterator &operator-- ()
iterator operator-- (int)
virtual Value operator[] (difference_type n) const
iterator &operator+= (difference_type n)
iterator operator+ (difference_type n) const
iterator &operator-= (difference_type n)
iterator operator- (difference_type n) const
bool operator== (const iterator &that) const
```

```
bool operator!=(const iterator &that) const
bool operator<(const iterator &that) const
bool operator<=(const iterator &that) const
bool operator>(const iterator &that) const
bool operator>=(const iterator &that) const
```

Protected Attributes

Value **state**
namespace tests

Functions

```
template<class Iterator, class T>
void trivial_iterator_test(const Iterator i, const Iterator j, T val)

template<typename Iterator, typename T>
void input_iterator_test(Iterator i, T v1, T v2)

template<typename Iterator, typename T>
void readable_iterator_traversal_test(Iterator i1, T v, std::true_type)

template<typename Iterator, typename T>
void readable_iterator_traversal_test(const Iterator, T, std::false_type)

template<typename Iterator, typename T>
void readable_iterator_test(const Iterator i1, T v)

template<typename Iterator, typename T>
void forward_iterator_test(Iterator i, T v1, T v2)

template<typename Iterator, typename T>
void forward_readable_iterator_test(Iterator i, Iterator j, T val1, T val2)

template<typename Iterator, typename T>
void bidirectional_iterator_test(Iterator i, T v1, T v2)

template<typename Iterator, typename T>
void bidirectional_readable_iterator_test(Iterator i, T v1, T v2)

template<typename Iterator, typename TrueVals>
void random_access_iterator_test(Iterator i, int N, TrueVals vals)

template<typename Iterator, typename TrueVals>
void random_access_readable_iterator_test(Iterator i, int N, TrueVals vals)

template<typename Iterator, typename T>
void constant_lvalue_iterator_test(Iterator i, T v1)

template<typename Iterator, typename T>
void non_const_lvalue_iterator_test(Iterator i, T v1, T v2)
```

```

template<typename Iterator, typename ConstIteratorconst_nonconst_iterator_test (Iterator i, ConstIterator j)

template<typename Iterator, typename T>
void writable_iterator_traversal_test (Iterator il, T v, std::true_type)

template<class Iterator, class T>
void writable_iterator_traversal_test (const Iterator, T, std::false_type)

template<class Iterator, class T>
void writable_iterator_test (Iterator i, T v, T v2)

struct dummy_type

```

Public Functions

```

dummy_type ()
dummy_type (dummy_constructor)
dummy_type (int x)
int foo () const
bool operator== (const dummy_type &d) const

```

Public Members

```

int x_

template<typename T>
class forward_iterator_archetype

```

Public Types

```

typedef forward_iterator_archetype self
typedef std::forward_iterator_tag iterator_category
typedef T value_type
typedef T const *pointer
typedef std::ptrdiff_t difference_type

```

Public Functions

```

forward_iterator_archetype ()
forward_iterator_archetype (forward_iterator_archetype const&)
self &operator= (const self&)
bool operator== (const self&) const
bool operator!= (const self&) const

```

```
reference operator* () const
self &operator++ ()
self operator++ (int)
```

Public Members

```
const typedef T& tests::forward_iterator_archetype::reference
template<typename T>
class input_iterator_archetype_no_proxy
```

Public Types

```
typedef std::input_iterator_tag iterator_category
typedef T value_type
typedef std::ptrdiff_t difference_type
```

Public Functions

```
input_iterator_archetype_no_proxy()
input_iterator_archetype_no_proxy (input_iterator_archetype_no_proxy const&)
self &operator= (const self&)
bool operator== (const self&) const
bool operator!= (const self&) const
reference operator* () const
self &operator++ ()
self operator++ (int)
```

Public Members

```
const typedef T& tests::input_iterator_archetype_no_proxy::reference
const typedef T* tests::input_iterator_archetype_no_proxy::pointer
```

Private Types

```
typedef input_iterator_archetype_no_proxy self
template<typename T>
class input_output_iterator_archetype
```

Public Types

```
typedef in_out_tag iterator_category
typedef T value_type
typedef std::ptrdiff_t difference_type
```

Public Functions

```
input_output_iterator_archetype()
self &operator=(const self&)
bool operator==(const self&) const
bool operator!=(const self&) const
reference operator*() const
self &operator++()
self operator++(int)
```

Public Members

```
const typedef T* tests::input_output_iterator_archetype::pointer
```

Private Types

```
typedef input_output_iterator_archetype self
struct reference
```

Public Functions

```
template<>
reference &operator=(const T&)

template<>
operator value_type()

template<class T>
class static_object
```

Public Static Functions

```
static T &get()
```

itt_notify

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_ITT_SYNC_CREATE (obj, type, name)
HPX_ITT_SYNC_RENAME (obj, name)
HPX_ITT_SYNC_PREPARE (obj)
HPX_ITT_SYNC_CANCEL (obj)
HPX_ITT_SYNC_ACQUIRED (obj)
HPX_ITT_SYNC_RELEASEING (obj)
HPX_ITT_SYNC_RELEASED (obj)
HPX_ITT_SYNC_DESTROY (obj)
HPX_ITT_STACK_CREATE (ctx)
HPX_ITT_STACK_CALLEE_ENTER (ctx)
HPX_ITT_STACK_CALLEE_LEAVE (ctx)
HPX_ITT_STACK_DESTROY (ctx)
HPX_ITT_FRAME_BEGIN (frame, id)
HPX_ITT_FRAME_END (frame, id)
HPX_ITT_MARK_CREATE (mark, name)
HPX_ITT_MARK_OFF (mark)
HPX_ITT_MARK (mark, parameter)
HPX_ITT_THREAD_SET_NAME (name)
HPX_ITT_THREAD_IGNORE ()
HPX_ITT_TASK_BEGIN (domain, name)
HPX_ITT_TASK_BEGIN_ID (domain, id, name)
HPX_ITT_TASK_END (domain)
HPX_ITT_DOMAIN_CREATE (name)
HPX_ITT_STRING_HANDLE_CREATE (name)
HPX_ITT_MAKE_ID (addr, extra)
HPX_ITT_ID_CREATE (domain, id)
HPX_ITT_ID_DESTROY (id)
HPX_ITT_HEAP_FUNCTION_CREATE (name, domain)
```

```

HPX_ITT_HEAP_ALLOCATE_BEGIN (f, size, initialized)
HPX_ITT_HEAP_ALLOCATE_END (f, addr, size, initialized)
HPX_ITT_HEAP_FREE_BEGIN (f, addr)
HPX_ITT_HEAP_FREE_END (f, addr)
HPX_ITT_HEAP_REALLOCATE_BEGIN (f, addr, new_size, initialized)
HPX_ITT_HEAP_REALLOCATE_END (f, addr, new_addr, new_size, initialized)
HPX_ITT_HEAP_INTERNAL_ACCESS_BEGIN ()
HPX_ITT_HEAP_INTERNAL_ACCESS_END ()
HPX_ITT_COUNTER_CREATE (name, domain)
HPX_ITT_COUNTER_CREATE_TYPED (name, domain, type)
HPX_ITT_COUNTER_SET_VALUE (id, value_ptr)
HPX_ITT_COUNTER_DESTROY (id)
HPX_ITT_METADATA_ADD (domain, id, key, data)

```

Typedefs

```
using __itt_heap_function = void*
```

Functions

```

constexpr void itt_sync_create (void*, const char*, const char*)
constexpr void itt_sync_rename (void*, const char*)
constexpr void itt_sync_prepare (void*)
constexpr void itt_sync_acquired (void*)
constexpr void itt_sync_cancel (void*)
constexpr void itt_sync_releasing (void*)
constexpr void itt_sync_released (void*)
constexpr void itt_sync_destroy (void*)
constexpr __itt_caller* itt_stack_create ()
constexpr void itt_stack_enter (__itt_caller*)
constexpr void itt_stack_leave (__itt_caller*)
constexpr void itt_stack_destroy (__itt_caller*)
constexpr void itt_frame_begin (__itt_domain const*, __itt_id*)
constexpr void itt_frame_end (__itt_domain const*, __itt_id*)
constexpr int itt_mark_create (char const*)
constexpr void itt_mark_off (int)
constexpr void itt_mark (int, char const*)
constexpr void itt_thread_set_name (char const*)

```

```
constexpr void itt_thread_ignore()

constexpr void itt_task_begin(__itt_domain const*, __itt_string_handle*)

constexpr void itt_task_begin(__itt_domain const*, __itt_id*, __itt_string_handle*)

constexpr void itt_task_end(__itt_domain const*)

constexpr __itt_domain *itt_domain_create(char const*)

constexpr __itt_string_handle *itt_string_handle_create(char const*)

constexpr __itt_id *itt_make_id(void*, unsigned long)

constexpr void itt_id_create(__itt_domain const*, __itt_id*)

constexpr void itt_id_destroy(__itt_id*)

constexpr __itt_heap_function itt_heap_function_create(const char*, const char*)

constexpr void itt_heap_allocate_begin(__itt_heap_function, std::size_t, int)

constexpr void itt_heap_allocate_end(__itt_heap_function, void**, std::size_t, int)

constexpr void itt_heap_free_begin(__itt_heap_function, void*)

constexpr void itt_heap_free_end(__itt_heap_function, void*)

constexpr void itt_heap_reallocate_begin(__itt_heap_function, void*, std::size_t, int)

constexpr void itt_heap_reallocate_end(__itt_heap_function, void*, void**, std::size_t, int)

constexpr void itt_heap_internal_access_begin()

constexpr void itt_heap_internal_access_end()

constexpr __itt_counter *itt_counter_create(char const*, char const*)

constexpr __itt_counter *itt_counter_create_typed(char const*, char const*, int)

constexpr void itt_counter_destroy(__itt_counter*)

constexpr void itt_counter_set_value(__itt_counter*, void*)

constexpr int itt_event_create(char const*, int)

constexpr int itt_event_start(int)

constexpr int itt_event_end(int)

constexpr void itt_metadata_add(__itt_domain*, __itt_id*, __itt_string_handle*, std::uint64_t
                               const&)

constexpr void itt_metadata_add(__itt_domain*, __itt_id*, __itt_string_handle*, double
                               const&)

constexpr void itt_metadata_add(__itt_domain*, __itt_id*, __itt_string_handle*, char const*)

constexpr void itt_metadata_add(__itt_domain*, __itt_id*, __itt_string_handle*, void
                               const*)

namespace hpx

namespace util

namespace itt
```

Functions

```
constexpr void event_tick(event const&)

struct caller_context
```

Public Functions

```
constexpr caller_context(stack_context&)

~caller_context()

struct counter
```

Public Functions

```
constexpr counter(char const*, char const*)

~counter()

struct domain
Subclassed by hpx::util::itt::thread_domain
```

Public Functions

```
HPX_NON_COPYABLE(domain)

constexpr domain(char const*)

domain()

struct event
```

Public Functions

```
constexpr event(char const*)

struct frame_context
```

Public Functions

```
constexpr frame_context(domain const&, id* = nullptr)

~frame_context()

struct heap_allocate
```

Public Functions

```
template<typename T>
constexpr heap_allocate(heap_function&, T**, std::size_t, int)

~heap_allocate()

struct heap_free
```

Public Functions

```
constexpr heap_free(heap_function&, void*)

~heap_free()

struct heap_function
```

Public Functions

```
constexpr heap_function(char const*, char const*)

~heap_function()

struct heap_internal_access
```

Public Functions

```
heap_internal_access()

~heap_internal_access()

struct id
```

Public Functions

```
constexpr id(domain const&, void*, unsigned long = 0)

~id()

struct mark_context
```

Public Functions

```
constexpr mark_context(char const*)

~mark_context()

struct mark_event
```

Public Functions

```
constexpr mark_event (event const&)

~mark_event ()

struct stack_context
```

Public Functions

```
stack_context()

~stack_context ()

struct string_handle
```

Public Functions

```
constexpr string_handle (char const* = nullptr)

struct task
```

Public Functions

```
constexpr task (domain const&, string_handle const&, std::uint64_t)

constexpr task (domain const&, string_handle const&)

~task ()

struct thread_domain : public hpx::util::itt::domain
```

Public Functions

```
HPX_NON_COPYABLE (thread_domain)

thread_domain()

struct undo_frame_context
```

Public Functions

```
constexpr undo_frame_context (frame_context const&)

~undo_frame_context ()

struct undo_mark_context
```

Public Functions

```
constexpr undo_mark_context(mark_context const&)
~undo_mark_context()
```

lcos_local

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```
struct and_gate : public hpx::lcos::local::base_and_gate<no_mutex>
```

Public Functions

```
and_gate(std::size_t count = 0)
and_gate(and_gate &&rhs)
and_gate &operator=(and_gate &&rhs)

template<typename Lock>
hpx::future<void> get_future(Lock &l, std::size_t count = std::size_t(-1), std::size_t
    *generation_value = nullptr, error_code &ec =
    hpx::throws)

template<typename Lock>
hpx::shared_future<void> get_shared_future(Lock &l, std::size_t count = std::size_t(-
    1), std::size_t *generation_value =
    nullptr, error_code &ec = hpx::throws)

template<typename Lock>
bool set(std::size_t which, Lock l, error_code &ec = throws)

template<typename Lock>
void synchronize(std::size_t generation_value, Lock &l, char const *function_name =
    "and_gate::synchronize", error_code &ec = throws)
```

Private Types

```
typedef base_and_gate<no_mutex> base_type

template<typename Mutex = lcos::local::spinlock>
struct base_and_gate
```

Public Functions

base_and_gate (*std::size_t count* = 0)

This constructor initializes the *base_and_gate* object from the the number of participants to synchronize the control flow with.

base_and_gate (*base_and_gate &&rhs*)

base_and_gate &**operator=** (*base_and_gate &&rhs*)

hpx::future<void> get_future (*std::size_t count* = *std::size_t(-1)*, *std::size_t *generation_value* = *nullptr*, *error_code &ec* = *hpx::throws*)

hpx::shared_future<void> get_shared_future (*std::size_t count* = *std::size_t(-1)*, *std::size_t *generation_value* = *nullptr*, *error_code &ec* = *hpx::throws*)

bool set (*std::size_t which*, *error_code &ec* = *throws*)

void synchronize (*std::size_t generation_value*, *char const *function_name* = "base_and_gate<>::synchronize", *error_code &ec* = *throws*)

Wait for the generational counter to reach the requested stage.

std::size_t next_generation()

std::size_t generation() const

Protected Types

typedef Mutex **mutex_type**

Protected Functions

bool trigger_conditions (*error_code &ec* = *throws*)

template<typename OuterLock>
hpx::future<void> get_future (*OuterLock &outer_lock*, *std::size_t count* = *std::size_t(-1)*, *std::size_t *generation_value* = *nullptr*, *error_code &ec* = *hpx::throws*)

get a future allowing to wait for the gate to fire

template<typename OuterLock>
hpx::shared_future<void> get_shared_future (*OuterLock &outer_lock*, *std::size_t count* = *std::size_t(-1)*, *std::size_t *generation_value* = *nullptr*, *error_code &ec* = *hpx::throws*)

get a shared future allowing to wait for the gate to fire

```
template<typename OuterLock>
bool set (std::size_t which, OuterLock outer_lock, error_code &ec = throws)
    Set the data which has to go into the segment which.

    bool test_condition (std::size_t generation_value)

template<typename Lock>
void synchronize (std::size_t generation_value, Lock &l, char const *function_name =
    "base_and_gate<>::synchronize", error_code &ec = throws)

template<typename OuterLock, typename Lock>
void init_locked (OuterLock &outer_lock, Lock &l, std::size_t count, error_code &ec =
    throws)
```

Private Types

```
typedef std::list<conditional_trigger*> condition_list_type
```

Private Members

```
mutex_type mtx_
boost::dynamic_bitset received_segments_
lcos::local::promise<void> promise_
std::size_t generation_
condition_list_type conditions_

struct manage_condition
```

Public Functions

```
template<>
manage_condition (base_and_gate &gate, conditional_trigger &cond)

template<>
~manage_condition ()

template<typename Condition>
hpx::future<void> get_future (Condition &&func, error_code &ec = hpx::throws)
```

Public Members

```
template<>
base_and_gate &this_

template<>
condition_list_type::iterator it_

namespace hpx
```



```
namespace lcos
```

```
namespace local
```

```
template<typename T>
class channel
```

Public Types

```
template<>
using value_type = T
```

Public Functions

```
channel()
```

Private Types

```
template<>
using base_type = detail::channel_base<T>
```

Friends

```
friend hpx::lcos::local::channel_iterator< T >
friend hpx::lcos::local::receive_channel< T >
friend hpx::lcos::local::send_channel< T >
```

```
template<>
class channel<void> : protected hpx::lcos::local::detail::channel_base<void>
```

Public Types

```
template<>
using value_type = void
```

Public Functions

```
channel()
```

Private Types

```
template<>
using base_type = detail::channel_base<void>
```

Friends

```
friend hpx::lcos::local::channel_iterator< void >
friend hpx::lcos::local::receive_channel< void >
friend hpx::lcos::local::send_channel< void >

template<typename T>
class channel_async_iterator : public hpx::util::iterator_facade<channel_async_iterator<T>, hpx::future<
```

Public Functions

```
channel_async_iterator()
channel_async_iterator(detail::channel_base<T> const *c)
```

Private Types

```
template<>
using base_type = hpx::util::iterator_facade<channel_async_iterator<T>, hpx::future<T>, std::input_iterator
```

Private Functions

```
std::pair<hpx::future<T>, bool> get_checked() const
bool equal(channel_async_iterator const &rhs) const
void increment()
base_type::reference dereference() const
```

Private Members

```
hpx::intrusive_ptr<detail::channel_impl_base<T>> channel_
std::pair<hpx::future<T>, bool> data_
```

Friends

```
friend hpx::lcos::local::hpx::util::iterator_core_access
template<typename T>
class channel_iterator : public hpx::util::iterator_facade<channel_iterator<T>, T const, std::input_iterator
```

Public Functions

```
channel_iterator()  
channel_iterator(detail::channel_base<T> const *c)  
channel_iterator(receive_channel<T> const *c)
```

Private Types

```
template<>  
using base_type = hpx::util::iterator_facade<channel_iterator<T>, T const, std::input_iterator_tag>
```

Private Functions

```
std::pair<T, bool> get_checked() const  
bool equal(channel_iterator const &rhs) const  
void increment()  
base_type::reference dereference() const
```

Private Members

```
hpx::intrusive_ptr<detail::channel_impl_base<T>> channel_  
std::pair<T, bool> data_
```

Friends

```
friend hpx::lcos::local::hpx::util::iterator_core_access  
template<>  
class channel_iterator<void> : public hpx::util::iterator_facade<channel_iterator<void>, util::unused_type
```

Public Functions

```
channel_iterator()  
channel_iterator(detail::channel_base<void> const *c)  
channel_iterator(receive_channel<void> const *c)
```

Private Types

```
template<>
using base_type = hpx::util::iterator_facade<channel_iterator<void>, util::unused_typeconst, std::input_i
```

Private Functions

```
bool get_checked()
bool equal(channel_iteratorconst &rhs) const
void increment()
base_type::reference dereference() const
```

Private Members

```
hpx::intrusive_ptr<detail::channel_base<util::unused_type>> channel_
bool data_
```

Friends

```
friend hpx::lcos::local::hpx::util::iterator_core_access
template<typename T>
class one_element_channel
```

Public Types

```
template<>
using value_type = T
```

Public Functions

```
one_element_channel()
```

Private Types

```
template<>
using base_type = detail::channel_base<T>
```

Friends

```
friend hpx::lcos::local::channel_iterator< T >
friend hpx::lcos::local::receive_channel< T >
friend hpx::lcos::local::send_channel< T >

template<>
class one_element_channel<void> : protected hpx::lcos::local::detail::channel_base<void>
```

Public Types

```
template<>
using value_type = void
```

Public Functions

```
one_element_channel()
```

Private Types

```
template<>
using base_type = detail::channel_base<void>
```

Friends

```
friend hpx::lcos::local::channel_iterator< void >
friend hpx::lcos::local::receive_channel< void >
friend hpx::lcos::local::send_channel< void >

template<typename T>
class receive_channel
```

Public Functions

```
receive_channel(channel<T> const &c)
receive_channel(one_element_channel<T> const &c)
```

Private Types

```
template<>
using base_type = detail::channel_base<T>
```

Friends

```
friend hpx::lcos::local::channel_iterator< T >
friend hpx::lcos::local::send_channel< T >

template<>
class receive_channel<void> : protected hpx::lcos::local::detail::channel_base<void>
```

Public Functions

```
receive_channel(channel<void> const &c)
receive_channel(one_element_channel<void> const &c)
```

Private Types

```
template<>
using base_type = detail::channel_base<void>
```

Friends

```
friend hpx::lcos::local::channel_iterator< void >
friend hpx::lcos::local::send_channel< void >

template<typename T>
class send_channel
```

Public Functions

```
send_channel(channel<T> const &c)
send_channel(one_element_channel<T> const &c)
```

Private Types

```
template<>
using base_type = detail::channel_base<T>

template<>
class send_channel<void> : private hpx::lcos::local::detail::channel_base<void>
```

Public Functions

```
send_channel(channel<void> const &c)
send_channel(one_element_channel<void> const &c)
```

Private Types

```
template<>
using base_type = detail::channel_base<void>
```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

Functions

```
void run_guarded(guard &guard, detail::guard_function task)
```

Conceptually, a guard acts like a mutex on an asynchronous task. The mutex is locked before the task runs, and unlocked afterwards.

```
template<typename F, typename ...Args>
```

```
void run_guarded(guard &guard, F &&f, Args&&... args)
```

```
void run_guarded(guard_set &guards, detail::guard_function task)
```

Conceptually, a *guard_set* acts like a set of mutexes on an asynchronous task. The mutexes are locked before the task runs, and unlocked afterwards.

```
template<typename F, typename ...Args>
```

```
void run_guarded(guard_set &guards, F &&f, Args&&... args)
```

```
class guard : public hpx::lcos::local::detail::debug_object
```

Public Functions

```
guard()
```

```
~guard()
```

Public Members

```
detail::guard_atomic task
```

```
class guard_set : public hpx::lcos::local::detail::debug_object
```

Public Functions

```
guard_set()
~guard_set()
std::shared_ptr<guard> get (std::size_t i)
void add (std::shared_ptr<guard> const &guard_ptr)
std::size_t size ()
```

Private Functions

```
void sort ()
```

Private Members

```
std::vector<std::shared_ptr<guard>> guards
bool sorted
```

Friends

```
void run_guarded (guard_set &guards, detail::guard_function task)
```

Conceptually, a *guard_set* acts like a set of mutexes on an asynchronous task. The mutexes are locked before the task runs, and unlocked afterwards.

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```
struct conditional_trigger
```

Public Functions

```
conditional_trigger()
conditional_trigger (conditional_trigger &&rhs)
conditional_trigger &operator= (conditional_trigger &&rhs)

template<typename Condition>
hpx::future<void> get_future (Condition &&func, error_code &ec = hpx::throws)
    get a future allowing to wait for the trigger to fire

void reset ()
bool set (error_code &ec = throws)
    Trigger this object.
```

Private Members

```

lcos::local::promise<void> promise_
util::function_nonsig<bool ()> cond_

namespace hpx

namespace lcos

namespace local

template<typename T, typename Mutex = lcos::local::spinlock>
struct receive_buffer

```

Public Functions

```

receive_buffer()

receive_buffer(receive_buffer &&other)

~receive_buffer()

receive_buffer &operator=(receive_buffer &&other)

hpX::future<T> receive(std::size_t step)

bool try_receive(std::size_t step, hpX::future<T> *f = nullptr)

template<typename Lock = hpX::lcos::local::no_mutex>
void store_received(std::size_t step, T &&val, Lock *lock = nullptr)

bool empty() const

std::size_t cancel_waiting(std::exception_ptr const &e, bool force_delete_entries =
false)

```

Protected Types

```

typedef Mutex mutex_type
typedef hpX::lcos::local::promise<T> buffer_promise_type
typedef std::map<std::size_t, std::shared_ptr<entry_data>> buffer_map_type
typedef buffer_map_type::iterator iterator

```

Protected Functions

```
iterator get_buffer_entry(std::size_t step)
```

Private Members

```
mutex_type mtx_
buffer_map_type buffer_map_
struct entry_data
```

Public Functions

```
template<>
HPX_NON_COPYABLE(entry_data)

template<>
entry_data()

template<>
hpx::future<T> get_future()

template<typename Val>
void set_value(Val &&val)

template<>
bool cancel(std::exception_ptr const &e)
```

Public Members

```
template<>
buffer_promise_type promise_

template<>
bool can_be_deleted_

template<>
bool value_set_

struct erase_on_exit
```

Public Functions

```
template<>
erase_on_exit(buffer_map_type &buffer_map, iterator it)

template<>
~erase_on_exit()
```

Public Members

```
template<>
buffer_map_type &buffer_map_
template<>
iterator it_
template<typename Mutex>
struct receive_buffer<void, Mutex>
```

Public Functions

```
receive_buffer()
receive_buffer(receive_buffer &&other)
~receive_buffer()
receive_buffer &operator=(receive_buffer &&other)
hpx::future<void> receive(std::size_t step)
bool try_receive(std::size_t step, hpx::future<void> *f = nullptr)
template<typename Lock = hpx::lcos::local::no_mutex>
void store_received(std::size_t step, Lock *lock = nullptr)
bool empty() const
std::size_t cancel_waiting(std::exception_ptr const &e, bool force_delete_entries =
false)
```

Protected Types

```
typedef Mutex mutex_type
typedef hpx::lcos::local::promise<void> buffer_promise_type
typedef std::map<std::size_t, std::shared_ptr<entry_data>> buffer_map_type
typedef buffer_map_type::iterator iterator
```

Protected Functions

```
iterator get_buffer_entry(std::size_t step)
```

Private Members

```
mutex_type mtx_
buffer_map_type buffer_map_
template<>
struct entry_data
```

Public Functions

```
template<>
HPX_NON_COPYABLE (entry_data)
template<>
entry_data ()
template<>
hpx::future<void> get_future ()
template<>
void set_value ()
template<>
bool cancel (std::exception_ptr const &e)
```

Public Members

```
template<>
buffer.promise_type promise_
template<>
bool can_be_deleted_
template<>
bool value_set_
template<>
struct erase_on_exit
```

Public Functions

```
template<>
erase_on_exit (buffer_map_type &buffer_map, iterator it)
template<>
~erase_on_exit ()
```

Public Members

```
template<>
buffer_map_type &buffer_map_
template<>
iterator it_

namespace hpx

namespace lcos

namespace local

template<typename Mutex = lcos::local::spinlock>
struct base_trigger
```

Public Functions

```
base_trigger()
base_trigger(base_trigger &&rhs)
base_trigger &operator=(base_trigger &&rhs)

hpx::future<void> get_future (std::size_t *generation_value = nullptr, error_code &ec =
                               hpx::throws)
    get a future allowing to wait for the trigger to fire

bool set (error_code &ec = throws)
    Trigger this object.

void synchronize (std::size_t generation_value, char const *function_name = "trigger::synchronize", error_code &ec = throws)
    Wait for the generational counter to reach the requested stage.

std::size_t next_generation()

std::size_t generation () const
```

Protected Types

```
typedef Mutex mutex_type
```

Protected Functions

```
bool trigger_conditions (error_code &ec = throws)  
  
template<typename Lock>  
void synchronize (std::size_t generation_value, Lock &l, char const *function_name = "trigger::synchronize", error_code &ec = throws)
```

Private Types

```
typedef std::list<conditional_trigger*> condition_list_type
```

Private Functions

```
bool test_condition (std::size_t generation_value)
```

Private Members

```
mutex_type mtx_  
lcos::local::promise<void> promise_  
std::size_t generation_  
condition_list_type conditions_  
struct manage_condition
```

Public Functions

```
template<>  
manage_condition (base_trigger &gate, conditional_trigger &cond)  
  
template<>  
~manage_condition ()  
  
template<typename Condition>  
hpx::future<void> get_future (Condition &&func, error_code &ec = hpx::throws)
```

Public Members

```
template<>  
base_trigger &this_  
  
template<>  
condition_list_type::iterator it_  
  
struct trigger : public hpx::lcos::local::base_trigger<no_mutex>
```

Public Functions

```
trigger()
trigger (trigger &&rhs)
trigger &operator= (trigger &&rhs)

template<typename Lock>
void synchronize (std::size_t generation_value, Lock &l, char const *function_name =
    "trigger::synchronize", error_code &ec = throws)
```

Private Types

```
typedef base_trigger<no_mutex> base_type
```

logging

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

```
namespace logging
```

Enums

```
enum level
```

Handling levels - classes that can hold and/or deal with levels.

- filters and level holders

By default we have these levels:

```
- debug (smallest level),
- info,
- warning ,
- error ,
- fatal (highest level)
```

Depending on which level is enabled for your application, some messages will reach the log: those messages having at least that level. For instance, if info level is enabled, all logged messages will reach the log. If warning level is enabled, all messages are logged, but the warnings. If debug level is enabled, messages that have levels debug, error, fatal will be logged.

Values:

```
disable_all = static_cast<unsigned int>(-1)
enable_all = 0
debug = 1000
info = 2000
```

```
warning = 3000
error = 4000
fatal = 5000
always = 6000
```

Functions

```
void format_value (std::ostream &os, boost::string_ref spec, level value)
```

Include this file when you're using the logging lib, but don't necessarily want to use formatters and destinations. If you want to use formatters and destinations, then you can include this one instead:

```
#include <hpx/logging/format.hpp>
```

```
namespace hpx
```

```
namespace util
```

```
namespace logging
```

```
namespace destination
```

Destination is a manipulator. It contains a place where the message, after being formatted, is to be written to.

Some viable destinations are : *the console, a file, a socket, etc.*

```
struct manipulator
```

```
#include <manipulator.hpp> What to use as base class, for your destination classes.
```

```
Subclassed by hpx::util::logging::destination::cerr, hpx::util::logging::destination::cout,
hpx::util::logging::destination::dbg_window, hpx::util::logging::destination::file,
hpx::util::logging::destination::stream
```

Public Functions

```
virtual void operator() (message const&) = 0
```

```
virtual void configure (std::string const&)
```

Override this if you want to allow configuration through scripting.

That is, this allows configuration of your manipulator at run-time.

```
virtual ~manipulator()
```

Protected Functions

manipulator()

namespace formatter

Formatter is a manipulator. It allows you to format the message before writing it to the destination(s)

Examples of formatters are : prepend the time, prepend high-precision time, prepend the index of the message, etc.

struct manipulator

#include <manipulator.hpp> What to use as base class, for your formatter classes.

Subclassed by *hpx::util::logging::formatter::high_precision_time, hpx::util::logging::formatter::idx, hpx::util::logging::formatter::thread_id*

Public Functions

virtual void operator() (std::ostream&) const = 0

virtual void configure (std::string const&)

Override this if you want to allow configuration through scripting.

That is, this allows configuration of your manipulator at run-time.

virtual ~manipulator()

Protected Functions

manipulator()

Friends

void format_value (std::ostream &os, boost::string_ref, manipulator const &value)

namespace hpx

namespace util

namespace logging

class message

#include <message.hpp> Optimizes the formatting for prepending and/or appending strings to the original message.

It keeps all the modified message in one string. Useful if some formatter needs to access the whole string at once.

reserve() - the size that is reserved for prepending (similar to string::reserve function)

Note : as strings are prepended, reserve() shrinks.

Public Functions

```
message()

message (std::stringstream msg)
Parameters
• msg: - the message that is originally cached

message (message &&other)

template<typename T>
message &operator<< (T &&v)

template<typename ...Args>
message &format (boost::string_ref format_str, Args const&... args)

std::string const &full_string () const
    returns the full string

bool empty () const
```

Private Members

```
std::stringstream m_str
bool m_full_msg_computed
std::string m_full_msg
```

Friends

```
std::ostream &operator<< (std::ostream &os, message const &value)

namespace hpx

namespace util

namespace logging
```

namespace destination

Destination is a manipulator. It contains a place where the message, after being formatted, is to be written to.

Some viable destinations are : *the console, a file*, a socket, etc.

```
struct cerr : public hpx::util::logging::destination::manipulator
    #include <destinations.hpp> Writes the string to cerr.
```

Public Functions

`~cerr()`

Public Static Functions

`static std::unique_ptr<cerr> make()`

Protected Functions

`cerr()`

`struct cout : public hpx::util::logging::destination::manipulator`
`#include <destinations.hpp>` Writes the string to console.

Public Functions

`~cout()`

Public Static Functions

`static std::unique_ptr<cout> make()`

Protected Functions

`cout()`

`struct dbg_window : public hpx::util::logging::destination::manipulator`
`#include <destinations.hpp>` Writes the string to output debug window.

For non-Windows systems, this is the console.

Public Functions

`~dbg_window()`

Public Static Functions

`static std::unique_ptr<dbg_window> make()`

Protected Functions

```
dbg_window()  
struct file : public hpx::util::logging::destination::manipulator  
#include <destinations.hpp> Writes the string to a file.
```

Public Functions

```
~file()
```

Public Static Functions

```
static std::unique_ptr<file> make (std::string const &file_name, file_settings set = {})  
constructs the file destination
```

Parameters

- file_name: name of the file
- set: [optional] file settings - see *file_settings* class, and *dealing_with_flags*

Protected Functions

```
file (std::string const &file_name, file_settings set)
```

Protected Attributes

```
std::string name  
file_settings settings  
struct file_settings  
#include <destinations.hpp> settings for when constructing a file class. To see how it's used,  
see dealing_with_flags.
```

Public Functions

```
file_settings()
```

Public Members

```
bool flush_each_time : 1  
if true (default), flushes after each write  
bool initial_overwrite : 1  
bool do_append : 1  
std::ios_base::openmode extra_flags  
just in case you have some extra flags to pass, when opening the file
```

```
struct stream : public hpx::util::logging::destination::manipulator
#include <destinations.hpp> writes to stream.
```

Note : The stream must outlive this object! Or, *clear()* the stream, before the stream is deleted.

Public Functions

```
~stream()
void set_stream(std::ostream *stream_ptr)
    resets the stream. Further output will be written to this stream
void clear()
    clears the stream. Further output will be ignored
```

Public Static Functions

```
static std::unique_ptr<stream> make (std::ostream *stream_ptr)
```

Protected Functions

```
stream (std::ostream *stream_ptr)
```

Protected Attributes

```
std::ostream *ptr
```

```
namespace hpx
```

```
namespace util
```

```
namespace logging
```

namespace formatter

Formatter is a manipulator. It allows you to format the message before writing it to the destination(s)

Examples of formatters are : prepend the time, prepend high-precision time, prepend the index of the message, etc.

```
struct high_precision_time : public hpx::util::logging::formatter::manipulator
#include <formatters.hpp> Prefixes the message with a high-precision time (. You pass the
format string at construction.
```

```
#include <hpx/logging/format/formatter/high_precision_time.hpp>
```

Internally, it uses hpx::util::date_time::microsec_time_clock. So, our precision matches this class.

The format can contain escape sequences: \$dd - day, 2 digits \$MM - month, 2 digits \$yy - year, 2 digits \$yyyy - year, 4 digits \$hh - hour, 2 digits \$mm - minute, 2 digits \$ss - second, 2 digits \$mili - milliseconds \$micro - microseconds (if the high precision clock allows; otherwise, it pads zeros) \$nano - nanoseconds (if the high precision clock allows; otherwise, it pads zeros)

Example:

```
high_precision_time("$mm:$ss:$micro");
```

Parameters

- convert: [optional] In case there needs to be a conversion between std::(w)string and the string that holds your logged message. See convert_format.

Public Functions

```
~high_precision_time()
```

Public Static Functions

```
static std::unique_ptr<high_precision_time> make(std::string const &format)
```

Protected Functions

```
high_precision_time(std::string const &format)
```

```
struct idx : public hpx::util::logging::formatter::manipulator  
#include <formatters.hpp> prefixes each message with an index.
```

Example:

```
L_ << "my message";  
L_ << "my 2nd message";
```

This will output something similar to:

```
[1] my message  
[2] my 2nd message
```

Public Functions

`~idx()`

Public Static Functions

`static std::unique_ptr<idx> make()`

Protected Functions

`idx()`

`struct thread_id : public hpx::util::logging::formatter::manipulator`
`#include <formatters.hpp>` Writes the `thread_id` to the log.

Parameters

- `convert`: [optional] In case there needs to be a conversion between `std::(w)string` and the string that holds your logged message. See `convert_format`.

Public Functions

`~thread_id()`

Public Static Functions

`static std::unique_ptr<thread_id> make()`

Protected Functions

`thread_id()`

`namespace hpx`

`namespace util`

`namespace logging`

`namespace writer`

`struct named_write`

`#include <named_write.hpp>` Composed of a named formatter and a named destinations. Thus, you can specify the formatting and destinations as strings.

`#include <hpx/logging/format/named_write.hpp>`

Contains a very easy interface for using formatters and destinations:

- at construction, specify 2 params: the formatter string and the destinations string
- Setting the formatters and destinations to write to is extremely simple:

```
// Set the formatters (first param) and destinatins (second step) in
// one step
g_l() ->writer().write("%time%($hh:$mm.$ss.$mili) [%idx%] |\n",
    "cout file(out.txt) debug");

// set the formatter(s)
g_l() ->writer().format("%time%($hh:$mm.$ss.$mili) [%idx%] |\n");

// set the destination(s)
g_l() ->writer().destination("cout file(out.txt) debug");
```

Public Functions

named_write()

void format (std::string const &format_str)

sets the format string: what should be before, and what after the original message, separated by “|”

Example: “[%idx%] |\n” - this writes “[%idx%] ” before the message, and “\n” after the message

If “|” is not present, the whole message is prepended to the message

void destination (std::string const &destination_str)

sets the destinations string - where should logged messages be outputted

void write (std::string const &format_str, std::string const &destination_str)

Specifies the formats and destinations in one step.

void operator () (message const &msg) const

template<typename **Formatter**>

void set_formatter (std::string const &name, Formatter fmt)

Replaces a formatter from the named formatter.

You can use this, for instance, when you want to share a formatter between multiple named writers.

template<typename **Formatter**, typename ...**Args**>

void set_formatter (std::string const &name, Args&&... args)

template<typename **Destination**>

void set_destination (std::string const &name, Destination dest)

Replaces a destination from the named destination.

You can use this, for instance, when you want to share a destination between multiple named writers.

template<typename **Destination**, typename ...**Args**>

void set_destination (std::string const &name, Args&&... args)

Private Functions

```
void configure_formatter(std::string const &format)  
void configure_destination(std::string const &format)
```

Private Members

```
detail::named_formatters m_format  
detail::named_destinations m_destination  
std::string m_format_str  
std::string m_destination_str
```

Defines

```
LAGAS_(lvl)  
LPT_(lvl)  
LTIM_(lvl)  
LPROGRESS_  
LHPX_(lvl, cat)  
LAPP_(lvl)  
LDEB_  
LTM_(lvl)  
LRT_(lvl)  
LOSH_(lvl)  
LERR_(lvl)  
LLCO_(lvl)  
LPCS_(lvl)  
LAS_(lvl)  
LBT_(lvl)  
LFATAL_  
LAGAS_CONSOLE_(lvl)  
LPT_CONSOLE_(lvl)  
LTIM_CONSOLE_(lvl)  
LHPX_CONSOLE_(lvl)  
LAPP_CONSOLE_(lvl)  
LDEB_CONSOLE_  
LAGAS_ENABLED(lvl)  
LPT_ENABLED(lvl)
```

```
LTIM_ENABLED (lvl)
LHPX_ENABLED (lvl)
LAPP_ENABLED (lvl)
LDEB_ENABLED
```

Functions

```
template<typename T>
bootstrap_logging const &operator<< (bootstrap_logging const &l, T&&)
```

Variables

```
constexpr bootstrap_logging lbt_
struct bootstrap_logging
```

Public Functions

```
constexpr bootstrap_logging ()
namespace hpx
```

Enums

```
enum logging_destination
Values:
destination_hpx = 0
destination_timing = 1
destination_agas = 2
destination_parcel = 3
destination_app = 4
destination_debuglog = 5
```

memory

See *Public API* for a list of names and headers that are part of the public HPX API.

```
template<typename T>
struct hash<hpx::memory::intrusive_ptr<T>>
```

Public Types

```
template<>
using result_type = std::size_t
```

Public Functions

```
result_type operator() (hpx::memory::intrusive_ptr<T> const &p) const  
namespace hpx
```

```
namespace memory
```

Functions

```
template<typename T, typename U>
constexpr bool operator== (intrusive_ptr<T> const &a, intrusive_ptr<U> const &b)  
  

template<typename T, typename U>
constexpr bool operator!= (intrusive_ptr<T> const &a, intrusive_ptr<U> const &b)  
  

template<typename T, typename U>
constexpr bool operator== (intrusive_ptr<T> const &a, U *b)  
  

template<typename T, typename U>
constexpr bool operator!= (intrusive_ptr<T> const &a, U *b)  
  

template<typename T, typename U>
constexpr bool operator== (T *a, intrusive_ptr<U> const &b)  
  

template<typename T, typename U>
constexpr bool operator!= (T *a, intrusive_ptr<U> const &b)  
  

template<typename T>
constexpr bool operator== (intrusive_ptr<T> const &p, std::nullptr_t)  
  

template<typename T>
constexpr bool operator== (std::nullptr_t, intrusive_ptr<T> const &p)  
  

template<typename T>
constexpr bool operator!= (intrusive_ptr<T> const &p, std::nullptr_t)  
  

template<typename T>
constexpr bool operator!= (std::nullptr_t, intrusive_ptr<T> const &p)  
  

template<typename T>
constexpr bool operator< (intrusive_ptr<T> const &a, intrusive_ptr<T> const &b)  
  

template<typename T>
void swap (intrusive_ptr<T> &lhs, intrusive_ptr<T> &rhs)  
  

template<typename T>
constexpr T *get_pointer (intrusive_ptr<T> const &p)  
  

template<typename T, typename U>
```

```
constexpr intrusive_ptr<T> static_pointer_cast (intrusive_ptr<U> const &p)

template<typename T, typename U>
constexpr intrusive_ptr<T> const_pointer_cast (intrusive_ptr<U> const &p)

template<typename T, typename U>
intrusive_ptr<T> dynamic_pointer_cast (intrusive_ptr<U> const &p)

template<typename T, typename U>
constexpr intrusive_ptr<T> static_pointer_cast (intrusive_ptr<U> &&p)

template<typename T, typename U>
constexpr intrusive_ptr<T> const_pointer_cast (intrusive_ptr<U> &&p)

template<typename T, typename U>
intrusive_ptr<T> dynamic_pointer_cast (intrusive_ptr<U> &&p)

template<typename Y>
std::ostream &operator<< (std::ostream &os, intrusive_ptr<Y> const &p)

template<typename T>
class intrusive_ptr
```

Public Types

```
template<>
using element_type = T
```

Public Functions

```
constexpr intrusive_ptr()

intrusive_ptr (T *p, bool add_ref = true)

template<typename U, typename Enable = std::enable_if_t<memory::detail::sp_convertible_v<U, T>>>
intrusive_ptr (intrusive_ptr<U> const &rhs)

intrusive_ptr (intrusive_ptr const &rhs)

~intrusive_ptr()

template<typename U>
intrusive_ptr &operator= (intrusive_ptr<U> const &rhs)

constexpr intrusive_ptr (intrusive_ptr &&rhs)

intrusive_ptr &operator= (intrusive_ptr &&rhs)

template<typename U, typename Enable = std::enable_if_t<memory::detail::sp_convertible_v<U, T>>>
constexpr intrusive_ptr (intrusive_ptr<U> &&rhs)

template<typename U>
intrusive_ptr &operator= (intrusive_ptr<U> &&rhs)

intrusive_ptr &operator= (intrusive_ptr const &rhs)

intrusive_ptr &operator= (T *rhs)
```

```

void reset ()
void reset (T *rhs)
void reset (T *rhs, bool add_ref)
constexpr T *get () const
constexpr T *detach ()
T &operator* () const
T *operator-> () const
constexpr operator bool () const
constexpr void swap (intrusive_ptr &rhs)

```

Private Types

```

template<>
using this_type = intrusive_ptr

```

Private Members

```
T *px = nullptr
```

Friends

```

friend hpx::memory::intrusive_ptr
namespace std

```

```

template<typename T>
struct hash<hpx::memory::intrusive_ptr<T>>

```

Public Types

```

template<>
using result_type = std::size_t

```

Public Functions

```

result_type operator() (hpx::memory::intrusive_ptr<T> const &p) const
namespace hpx
namespace serialization

```

Functions

```
template<typename T>
void load(input_archive &ar, hpx::intrusive_ptr<T> &ptr, unsigned)

template<typename T>
void save(output_archive &ar, hpx::intrusive_ptr<T> const &ptr, unsigned)
```

mpi_base

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
namespace hpx
```

```
namespace util
```

```
struct mpi_environment
```

Public Static Functions

```
static bool check_mpi_environment(runtime_configuration const &cfg)
```

pack_traversal

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename Mapper, typename... T><unspecified> hpx::util::map_pack(Mapper &
```

Maps the pack with the given mapper.

This function tries to visit all plain elements which may be wrapped in:

- homogeneous containers (`std::vector`, `std::list`)
- heterogeneous containers (`hpx::tuple`, `std::pair`, `std::array`) and re-assembles the pack with the result of the mapper. Mapping from one type to a different one is supported.

Elements that aren't accepted by the mapper are routed through and preserved through the hierarchy.

```
// Maps all integers to floats
map_pack([](int value) {
    return float(value);
},
1, hpx::make_tuple(2, std::vector<int>{3, 4}), 5);
```

Return The mapped element or in case the pack contains multiple elements, the pack is wrapped into a `hpx::tuple`.

Exceptions

- `std::exception`: like objects which are thrown by an invocation to the mapper.

Parameters

- `mapper`: A callable object, which accept an arbitrary type and maps it to another type or the same one.
- `pack`: An arbitrary variadic pack which may contain any type.

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename Visitor, typename ...T>
auto traverse_pack_async (Visitor &&visitor, T&&... pack)
```

Traverses the pack with the given visitor in an asynchronous way.

This function works in the same way as `traverse_pack`, however, we are able to suspend and continue the traversal at later time. Thus we require a visitor callable object which provides three `operator()` overloads as depicted by the code sample below:

```
struct my_async_visitor
{
    template <typename T>
    bool operator() (async_traverse_visit_tag, T&& element)
    {
        return true;
    }

    template <typename T, typename N>
    void operator() (async_traverse_detach_tag, T&& element, N&& next)
    {
    }

    template <typename T>
    void operator() (async_traverse_complete_tag, T&& pack)
    {
    }
};
```

See `traverse_pack` for a detailed description about the traversal behavior and capabilities.

Return A `hpx::intrusive_ptr` that references an instance of the given visitor object.

Parameters

- `visitor`: A visitor object which provides the three `operator()` overloads that were described above. Additionally the visitor must be compatible for referencing it from a `hpx::intrusive_ptr`. The visitor should must have a virtual destructor!
- `pack`: The arbitrary parameter pack which is traversed asynchronously. Nested objects inside containers and tuple like types are traversed recursively.

```
template<typename Allocator, typename Visitor, typename ...T>
```

```
auto traverse_pack_async_allocator(Allocator const &alloc, Visitor &&visitor,
                                T&&... pack)
```

Traverses the pack with the given visitor in an asynchronous way.

This function works in the same way as `traverse_pack`, however, we are able to suspend and continue the traversal at later time. Thus we require a visitor callable object which provides three `operator()` overloads as depicted by the code sample below:

```
struct my_async_visitor
{
    template <typename T>
    bool operator() (async_traverse_visit_tag, T&& element)
    {
        return true;
    }

    template <typename T, typename N>
    void operator() (async_traverse_detach_tag, T&& element, N&& next)
    {
    }

    template <typename T>
    void operator() (async_traverse_complete_tag, T&& pack)
    {
    }
};
```

See `traverse_pack` for a detailed description about the traversal behavior and capabilities.

Return A `hpx::intrusive_ptr` that references an instance of the given visitor object.

Parameters

- `visitor`: A visitor object which provides the three `operator()` overloads that were described above. Additionally the visitor must be compatible for referencing it from a `hpx::intrusive_ptr`. The visitor should must have a virtual destructor!
- `pack`: The arbitrary parameter pack which is traversed asynchronously. Nested objects inside containers and tuple like types are traversed recursively.
- `alloc`: Allocator instance to use to create the traversal frame.

```
namespace hpx
```

Functions

```
template<typename ...Args>
auto unwrap(Args&&... args)
```

A helper function for retrieving the actual result of any `hpx::future` like type which is wrapped in an arbitrary way.

Unwraps the given pack of arguments, so that any `hpx::future` object is replaced by its future result type in the argument pack:

- `hpx::future<int> -> int`
- `hpx::future<std::vector<float>> -> std::vector<float>`
- `std::vector<future<float>> -> std::vector<float>`

The function is capable of unwrapping hpx::future like objects that are wrapped inside any container or tuple like type, see hpx::util::map_pack() for a detailed description about which surrounding types are supported. Non hpx::future like types are permitted as arguments and passed through.

```
// Single arguments
int i1 = hpx::unwrap(hpx::make_ready_future(0));

// Multiple arguments
hpx::tuple<int, int> i2 =
    hpx::unwrap(hpx::make_ready_future(1),
                hpx::make_ready_future(2));
```

Note This function unwraps the given arguments until the first traversed nested hpx::future which corresponds to an unwrapping depth of one. See hpx::unwrap_n() for a function which unwraps the given arguments to a particular depth or hpx::unwrap_all() that unwraps all future like objects recursively which are contained in the arguments.

Return Depending on the count of arguments this function returns a hpx::tuple containing the unwrapped arguments if multiple arguments are given. In case the function is called with a single argument, the argument is unwrapped and returned.

Parameters

- args: the arguments that are unwrapped which may contain any arbitrary future or non future type.

Exceptions

- std::exception: like objects in case any of the given wrapped hpx::future objects were resolved through an exception. See hpx::future::get() for details.

```
template<std::size_t Depth, typename ...Args>
auto unwrap_n(Args&&... args)
```

An alterntive version of hpx::unwrap(), which unwraps the given arguments to a certain depth of hpx::future like objects.

See `unwrap` for a detailed description.

Template Parameters

- Depth: The count of hpx::future like objects which are unwrapped maximally.

```
template<typename ...Args>
auto unwrap_all(Args&&... args)
```

An alterntive version of hpx::unwrap(), which unwraps the given arguments recursively so that all contained hpx::future like objects are replaced by their actual value.

See `hpx::unwrap()` for a detailed description.

```
template<typename T>
auto unwrapping(T &&callable)
```

Returns a callable object which unwraps its arguments upon invocation using the `hpx::unwrap()` function and then passes the result to the given callable object.

```
auto callable = hpx::unwrapping([](int left, int right) {
    return left + right;
});

int i1 = callable(hpx::make_ready_future(1),
                  hpx::make_ready_future(2));
```

See `hpx::unwrap()` for a detailed description.

Parameters

- `callable`: the callable object which is called with the result of the corresponding `unwrap` function.

```
template<std::size_t Depth, typename T>
auto unwrapping_n(T &&callable)
    Returns a callable object which unwraps its arguments upon invocation using the hpx::unwrap_n() function and then passes the result to the given callable object.
```

See `hpx::unwrapping()` for a detailed description.

```
template<typename T>
auto unwrapping_all(T &&callable)
    Returns a callable object which unwraps its arguments upon invocation using the hpx::unwrap_all() function and then passes the result to the given callable object.
```

See `hpx::unwrapping()` for a detailed description.

namespace functional

struct unwrap

```
#include <unwrap.hpp> A helper function object for functionally invoking hpx::unwrap. For more information please refer to its documentation.
```

struct unwrap_all

```
#include <unwrap.hpp> A helper function object for functionally invoking hpx::unwrap_all. For more information please refer to its documentation.
```

template<std::size_t Depth>

struct unwrap_n

```
#include <unwrap.hpp> A helper function object for functionally invoking hpx::unwrap_n. For more information please refer to its documentation.
```

namespace util

Functions

```
template<typename ...Args>
auto unwrap(Args&&... args)
```

```
template<std::size_t Depth, typename ...Args>
auto unwrap_n(Args&&... args)
```

```
template<typename ...Args>
auto unwrap_all(Args&&... args)
```

```
template<typename T>
auto unwrapping(T &&callable)

template<std::size_t Depth, typename T>
auto unwrapping_n(T &&callable)

template<typename T>
auto unwrapping_all(T &&callable)

namespace functional
```

Functions

```
struct hpx::util::functional::HPX_DEPRECATED_V(1, 7, "Please use hpx::functional
template<typename NewType, typename OldType, typename OldAllocator>
struct pack_traversal_rebind_container<NewType, std::vector<OldType, OldAllocator>>
```

Public Types

```
template<>
using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>
```

Public Static Functions

```
static std::vector<NewType, NewAllocator> call(std::vector<OldType, OldAllocator> const
&container)
```

```
template<typename NewType, typename OldType, typename OldAllocator>
struct pack_traversal_rebind_container<NewType, std::list<OldType, OldAllocator>>
```

Public Types

```
template<>
using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>
```

Public Static Functions

```
static std::list<NewType, NewAllocator> call(std::list<OldType, OldAllocator> const &con-
tainer)
```

```
template<typename NewType, typename OldType, std::size_t N>
struct pack_traversal_rebind_container<NewType, std::array<OldType, N>>
```

Public Static Functions

```
static std::array<NewType, N> call (std::array<OldType, N> const&)

namespace hpx

namespace traits

template<typename NewType, typename OldType, std::size_t N>
struct pack_traversal_rebind_container<NewType, std::array<OldType, N>>
```

Public Static Functions

```
static std::array<NewType, N> call (std::array<OldType, N> const&)

template<typename NewType, typename OldType, typename OldAllocator>
struct pack_traversal_rebind_container<NewType, std::list<OldType, OldAllocator>>
```

Public Types

```
template<>
using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>
```

Public Static Functions

```
static std::list<NewType, NewAllocator> call (std::list<OldType, OldAllocator> const
                                              &container)

template<typename NewType, typename OldType, typename OldAllocator>
struct pack_traversal_rebind_container<NewType, std::vector<OldType, OldAllocator>>
```

Public Types

```
template<>
using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>
```

Public Static Functions

```
static std::vector<NewType, NewAllocator> call (std::vector<OldType, OldAllocator>
                                                const &container)
```

plugin

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_PLUGIN_EXPORT_API
HPX_PLUGIN_API
HPX_PLUGIN_ARGUMENT_LIMIT
HPX_PLUGIN_SYMBOLS_PREFIX_DYNAMIC
HPX_PLUGIN_SYMBOLS_PREFIX
HPX_PLUGIN_SYMBOLS_PREFIX_DYNAMIC_STR
HPX_PLUGIN_SYMBOLS_PREFIX_STR
namespace hpx

namespace util

namespace plugin
```

Typedefs

```
template<typename T>
using shared_ptr = boost::shared_ptr<T>
```

Defines

```
HPX_HAS_DLOPEN
```

Defines

```
HPX_PLUGIN_NAME_2(name1, name2)
HPX_PLUGIN_NAME_3(name, base, cname)
HPX_PLUGIN_LIST_NAME_(prefix, name, base)
HPX_PLUGIN_EXPORTER_NAME_(prefix, name, base, cname)
HPX_PLUGIN_EXPORTER_INSTANCE_NAME_(prefix, name, base, cname)
HPX_PLUGIN_FORCE_LOAD_NAME_(prefix, name, base)
HPX_PLUGIN_LIST_NAME(name, base)
HPX_PLUGIN_EXPORTER_NAME(name, base, cname)
HPX_PLUGIN_EXPORTER_INSTANCE_NAME(name, base, cname)
HPX_PLUGIN_FORCE_LOAD_NAME(name, base)
HPX_PLUGIN_LIST_NAME_DYNAMIC(name, base)
```

```
HPX_PLUGIN_EXPORTER_NAME_DYNAMIC (name, base, cname)
HPX_PLUGIN_EXPORTER_INSTANCE_NAME_DYNAMIC (name, base, cname)
HPX_PLUGIN_FORCE_LOAD_NAME_DYNAMIC (name, base)
HPX_PLUGIN_EXPORT_ (prefix, name, BaseType, ActualType, actualname, classname)
HPX_PLUGIN_EXPORT (name, BaseType, ActualType, actualname, classname)
HPX_PLUGIN_EXPORT_DYNAMIC (name, BaseType, ActualType, actualname, classname)
HPX_PLUGIN_EXPORT_LIST_ (prefix, name, classname)
HPX_PLUGIN_EXPORT_LIST (name, classname)
HPX_PLUGIN_EXPORT_LIST_DYNAMIC (name, classname)

namespace hpx

namespace util

namespace plugin

template<class BasePlugin>
struct plugin_factory : public hpx::util::plugin::detail::plugin_factory_item<BasePlugin, detail::plugin_fac
```

Public Functions

```
plugin_factory (dll &d, std::string const &basename)
```

Private Types

```
template<>
using base_type = detail::plugin_factory_item<BasePlugin, detail::plugin_factory_item_base, typename v
template<class BasePlugin>
struct static_plugin_factory : public hpx::util::plugin::detail::static_plugin_factory_item<BasePlugin,
```

Public Functions

```
static_plugin_factory (get_plugins_list_type const &f)
```

Private Types

```
template<>
using base_type = detail::static_plugin_factory_item<BasePlugin, detail::static_plugin_factory_item_base, t
namespace hpx

namespace util
```

```
namespace plugin
```

```
template<typename Wrapped, typename ...Parameters>
struct plugin_wrapper : public hpx::util::plugin::detail::dll_handle_holder, public Wrapped
```

Public Functions

```
plugin_wrapper(dll_handle dll, Parameters... parameters)
```

```
namespace hpx
```

```
namespace util
```

```
namespace plugin
```

TypeDefs

```
using exported_plugins_type = std::map<std::string, hpx::any_nonser>
typedef exported_plugins_type* HPX_PLUGIN_API* hpx::util::plugin::get_plugins_list_np
typedef exported_plugins_type* HPX_PLUGIN_API hpx::util::plugin::get_plugins_list_np
using dll_handle = shared_ptr<get_plugins_list_np>

template<typename BasePlugin>
struct virtual_constructor
```

Public Types

```
template<>
using type = hpx::util::pack<>
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename Plugin, typename Enable = void>
struct plugin_config_data
```

Public Static Functions

```
static char const *call()
```

prefix

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_BASE_DIR_NAME  
HPX_DEFAULT_INI_PATH  
HPX_DEFAULT_INI_FILE  
HPX_DEFAULT_COMPONENT_PATH  
namespace hpx  
  
namespace util
```

Functions

```
void set_hpx_prefix(const char *prefix)  
char const *hpx_prefix()  
std::string find_prefix(std::string const &library = "hpx")  
std::string find_prefixes(std::string const &suffix, std::string const &library = "hpx")  
std::string get_executable_filename(char const *argv0 = nullptr)  
std::string get_executable_prefix(char const *argv0 = nullptr)
```

preprocessor

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

HPX_PP_CAT (*A, B*)

Concatenates the tokens *A* and *B* into a single token. Evaluates to *AB*

Parameters

- *A*: First token
- *B*: Second token

Defines

HPX_PP_EXPAND (X)

The HPX_PP_EXPAND macro performs a double macro-expansion on its argument. This macro can be used to produce a delayed preprocessor expansion.

Parameters

- X: Token to be expanded twice

Example:

```
#define MACRO(a, b, c) (a)(b)(c)
#define ARGS() (1, 2, 3)

HPX_PP_EXPAND(MACRO ARGS()) // expands to (1)(2)(3)
```

Defines

HPX_PP_IDENTITY (...)

Defines

HPX_PP_NARGS (...)

Expands to the number of arguments passed in

Example Usage:

```
HPX_PP_NARGS(hpx, pp, nargs)
HPX_PP_NARGS(hpx, pp)
HPX_PP_NARGS(hpx)
```

Parameters

- ...: The variadic number of arguments

Expands to:

```
3
2
1
```

Defines

HPX_PP_STRINGIZE (X)

The *HPX_PP_STRINGIZE* macro stringizes its argument after it has been expanded.

The passed argument X will expand to "X". Note that the stringizing operator (#) prevents arguments from expanding. This macro circumvents this shortcoming.

Parameters

- X: The text to be converted to a string literal

Defines

HPX_PP_STRIP_PARENS (*X*)

For any symbol *X*, this macro returns the same symbol from which potential outer parens have been removed. If no outer parens are found, this macro evaluates to *X* itself without error.

The original implementation of this macro is from Steven Watanbe as shown in <http://boost.2283326.n4.nabble.com/preprocessor-removing-parentheses-td2591973.html#a2591976>

```
HPX_PP_STRIP_PARENS (no_parens)
HPX_PP_STRIP_PARENS ( (with_parens) )
```

Example Usage:

Parameters

- *X*: Symbol to strip parens from

This produces the following output

```
no_parens
with_parens
```

program_options

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

```
namespace program_options

namespace command_line_style
```

Enums

enum style_t

Various possible styles of options.

There are “long” options, which start with “–” and “short”, which start with either “-” or “/”. Both kinds can be allowed or disallowed, see `allow_long` and `allow_short`. The allowed character for short options is also configurable.

Option’s value can be specified in the same token as name (“`-foo=bar`”), or in the next token.

It’s possible to introduce long options by the same character as short options, see `allow_long_disguise`.

Finally, guessing (specifying only prefix of option) and case insensitive processing are supported.

Values:

`allow_long = 1`

Allow “`-long_name`” style.

```

allow_short = allow_long << 1
    Allow “-<single character” style.

allow_dash_for_short = allow_short << 1
    Allow “-” in short options.

allow_slash_for_short = allow_dash_for_short << 1
    Allow “/” in short options.

long_allow_adjacent = allow_slash_for_short << 1
    Allow option parameter in the same token for long option, like in
    

--foo=10

long_allow_next = long_allow_adjacent << 1
    Allow option parameter in the next token for long options.

short_allow_adjacent = long_allow_next << 1
    Allow option parameter in the same token for short options.

short_allow_next = short_allow_adjacent << 1
    Allow option parameter in the next token for short options.

allow_sticky = short_allow_next << 1
    Allow to merge several short options together, so that “-s -k” become “-sk”. All of the options but last should accept no parameter. For example, if “-s” accept a parameter, then “k” will be taken as parameter, not another short option. Dos-style short options cannot be sticky.

allow_guessing = allow_sticky << 1
    Allow abbreviated spellings for long options, if they unambiguously identify long option. No long option name should be prefix of other long option name if guessing is in effect.

long_case_insensitive = allow_guessing << 1
    Ignore the difference in case for long options.

short_case_insensitive = long_case_insensitive << 1
    Ignore the difference in case for short options.

case_insensitive = (long_case_insensitive | short_case_insensitive)
    Ignore the difference in case for all options.

allow_long_disguise = short_case_insensitive << 1
    Allow long options with single option starting character, e.g -f○o=10

unix_style = (allow_short | short_allow_adjacent | short_allow_next | allow_long | long_allow_adjacent | long_allow_next)
    The more-or-less traditional unix style.

default_style = unix_style
    The default style.

namespace hpx

namespace program_options

```

Typedefs

```
using any = hpx::any_nonser
template<typename T>
using optional = hpx::util::optional<T>

namespace hpx

namespace program_options

class environment_iterator : public hpx::program_options::eof_iterator<environment_iterator, std::pair<std::string, std::string>>
```

Public Functions

```
environment_iterator(char **environment)
environment_iterator()
void get()
```

Private Members

```
char **m_environment

namespace hpx

namespace program_options
```

```
template<class Derived, class ValueType>
class eof_iterator : public util::iterator_facade<Derived, ValueType const, std::forward_iterator_tag>
#include <eof_iterator.hpp> The ‘eof_iterator’ class is useful for constructing forward iterators in cases where iterator extract data from some source and it’s easy to detect ‘eof’ – i.e. the situation where there’s no data. One apparent example is reading lines from a file.
```

Implementing such iterators using ‘iterator_facade’ directly would require to create class with three core operation, a couple of constructors. When using ‘eof_iterator’, the derived class should define only one method to get new value, plus a couple of constructors.

The basic idea is that iterator has ‘eof’ bit. Two iterators are equal only if both have their ‘eof’ bits set. The ‘get’ method either obtains the new value or sets the ‘eof’ bit.

Specifically, derived class should define:

1. A default constructor, which creates iterator with ‘eof’ bit set. The constructor body should call ‘found_eof’ method defined here.
2. Some other constructor. It should initialize some ‘data pointer’ used in iterator operation and then call ‘get’.
3. The ‘get’ method. It should operate this way:
 - look at some ‘data pointer’ to see if new element is available; if not, it should call ‘found_eof’.
 - extract new element and store it at location returned by the ‘value’ method.
 - advance the data pointer.

Essentially, the ‘get’ method has the functionality of both ‘increment’ and ‘dereference’. It’s very good for the cases where data extraction implicitly moves data pointer, like for stream operation.

Public Functions

```
eof_iterator()
```

Protected Functions

```
ValueType &value()
```

Returns the reference which should be used by derived class to store the next value.

```
void found_eof()
```

Should be called by derived class to indicate that it can’t produce next element.

Private Functions

```
void increment()
```

```
bool equal (const eof_iterator &other) const  
const ValueType &dereference() const
```

Private Members

```
bool m_at_eof
```

```
ValueType m_value
```

Friends

```
friend hpx::program_options::hpx::util::iterator_core_access  
namespace hpx  
  
namespace program_options
```

Functions

```
std::string strip_prefixes (const std::string &text)
```

```
class ambiguous_option : public hpx::program_options::error_with_no_option_name  
#include <errors.hpp> Class thrown when there's ambiguity among several possible options.
```

Public Functions

```
ambiguous_option (const std::vector<std::string> &xalternatives)
~ambiguous_option ()
const std::vector<std::string> &alternatives () const
```

Protected Functions

```
void substitute_placeholders (const std::string &error_template) const
    Makes all substitutions using the template
```

Private Members

```
std::vector<std::string> m_alternatives
class error : public logic_error
#include <errors.hpp> Base class for all errors in the library.
Subclassed by hpx::program_options::duplicate_option_error, hpx::program_options::error_with_option_name,
hpx::program_options::invalid_command_line_style,           hpx::program_options::reading_file,
hpx::program_options::too_many_positional_options_error
```

Public Functions

```
error (const std::string &xwhat)
class error_with_no_option_name : public hpx::program_options::error_with_option_name
#include <errors.hpp> Base class of un-parsable options, when the desired option cannot be identified.
It makes no sense to have an option name, when we can't match an option to the parameter
Having this a part of the error_with_option_name hierarchy makes error handling a lot easier, even if the name indicates some sort of conceptual dissonance!
Subclassed by hpx::program_options::ambiguous_option, hpx::program_options::unknown_option
```

Public Functions

```
error_with_no_option_name (const std::string &template_, const std::string &original_token = "")
void set_option_name (const std::string&)
    Does NOT set option name, because no option name makes sense
~error_with_no_option_name ()

class error_with_option_name : public hpx::program_options::error
#include <errors.hpp> Base class for most exceptions in the library.
Substitutes the values for the parameter name placeholders in the template to create the human readable error message
```

Placeholders are surrounded by % signs: example% Poor man's version of boost::format

If a parameter name is absent, perform default substitutions instead so ugly placeholders are never left in-place.

Options are displayed in “canonical” form This is the most unambiguous form of the *parsed* option name and would correspond to *option_description::format_name()* i.e. what is shown by *print_usage()*

The “canonical” form depends on whether the option is specified in short or long form, using dashes or slashes or without a prefix (from a configuration file)

Subclassed by	<i>hpx::program_options::error_with_no_option_name,</i>
	<i>hpx::program_options::invalid_syntax,</i>
	<i>hpx::program_options::multiple_occurrences,</i>
	<i>hpx::program_options::multiple_values,</i>
	<i>hpx::program_options::required_option,</i>
	<i>hpx::program_options::validation_error</i>

Public Functions

```
error_with_option_name(const std::string &template_, const std::string &option_name = "", const std::string &original_token = "", int option_style = 0)
```

```
~error_with_option_name()
```

gcc says that throw specification on dtor is loosened without this line

```
void set_substitute(const std::string &parameter_name, const std::string &value)
```

Substitute parameter_name->value to create the error message from the error template

```
void set_substitute_default(const std::string &parameter_name, const std::string &from, const std::string &to)
```

If the parameter is missing, then make the from->to substitution instead

```
void add_context(const std::string &option_name, const std::string &original_token, int option_style)
```

Add context to an exception

```
void set_prefix(int option_style)
```

```
virtual void set_option_name(const std::string &option_name)
```

Overridden in *error_with_no_option_name*

```
std::string get_option_name() const
```

```
void set_original_token(const std::string &original_token)
```

```
const char *what() const
```

Creates the error_message on the fly Currently a thin wrapper for *substitute_placeholders()*

Public Members

```
std::string m_error_template  
    template with placeholders
```

Protected Types

```
using string_pair = std::pair<std::string, std::string>
```

Protected Functions

```
virtual void substitute_placeholders (const std::string &error_template)  
    const
```

Makes all substitutions using the template

```
void replace_token (const std::string &from, const std::string &to) const
```

```
std::string get_canonical_option_name () const
```

Construct option name in accordance with the appropriate prefix style: i.e. long dash or short slash etc

```
std::string get_canonical_option_prefix () const
```

Protected Attributes

```
int m_option_style
```

can be 0 = no prefix (config file options) allow_long allow_dash_for_short allow_slash_for_short
allow_long_disguise

```
std::map<std::string, std::string> m_substitutions  
    substitutions from placeholders to values
```

```
std::map<std::string, string_pair> m_substitution_defaults
```

```
std::string m_message
```

Used to hold the error text returned by what()

```
class invalid_bool_value : public hpx::program_options::validation_error  
#include <errors.hpp> Class thrown if there is an invalid bool value given
```

Public Functions

```
invalid_bool_value (const std::string &value)
```

```
class invalid_command_line_style : public hpx::program_options::error  
#include <errors.hpp> Class thrown when there are programming error related to style
```

Public Functions

```
invalid_command_line_style(const std::string &msg)
class invalid_command_line_syntax: public hpx::program_options::invalid_syntax
#include <errors.hpp> Class thrown when there are syntax errors in given command line
```

Public Functions

```
invalid_command_line_syntax(kind_t kind, const std::string &option_name = "",
const std::string &original_token = "", int option_style = 0)
~invalid_command_line_syntax()

class invalid_config_file_syntax: public hpx::program_options::invalid_syntax
```

Public Functions

```
invalid_config_file_syntax(const std::string &invalid_line, kind_t kind)
~invalid_config_file_syntax()

std::string tokens() const
    Convenience functions for backwards compatibility

class invalid_option_value: public hpx::program_options::validation_error
#include <errors.hpp> Class thrown if there is an invalid option value given
```

Public Functions

```
invalid_option_value(const std::string &value)
invalid_option_value(const std::wstring &value)

class invalid_syntax: public hpx::program_options::error_with_option_name
#include <errors.hpp> Class thrown when there's syntax error either for command line or config file options. See derived children for concrete classes.

Subclassed by hpx::program_options::invalid_command_line_syntax,  

hpx::program_options::invalid_config_file_syntax
```

Public Types

```
enum kind_t
    Values:
        long_not_allowed=30
        long_adjacent_not_allowed
        short_adjacent_not_allowed
        empty_adjacent_parameter
        missing_parameter
```

```
extra_parameter
unrecognized_line
```

Public Functions

```
invalid_syntax(kind_t kind, const std::string &option_name = "", const std::string
&original_token = "", int option_style = 0)
~invalid_syntax()

kind_t kind() const
virtual std::string tokens() const
Convenience functions for backwards compatibility
```

Protected Functions

```
std::string get_template(kind_t kind)
Used to convert kind_t to a related error text
```

Protected Attributes

```
kind_t m_kind
```

```
class multiple_occurrences : public hpx::program_options::error_with_option_name
#include <errors.hpp> Class thrown when there are several occurrences of an option, but user called
a method which cannot return them all.
```

Public Functions

```
multiple_occurrences()
~multiple_occurrences()

class multiple_values : public hpx::program_options::error_with_option_name
#include <errors.hpp> Class thrown when there are several option values, but user called a method
which cannot return them all.
```

Public Functions

```
multiple_values()
~multiple_values()

class reading_file : public hpx::program_options::error
#include <errors.hpp> Class thrown if config file can not be read
```

Public Functions

```
reading_file(const char *filename)

class required_option:public hpx::program_options::error_with_option_name
    #include <errors.hpp> Class thrown when a required/mandatory option is missing
```

Public Functions

```
required_option(const std::string &option_name)

~required_option()

class too_many_positional_options_error:public hpx::program_options::error
    #include <errors.hpp> Class thrown when there are too many positional options. This is a programming error.
```

Public Functions

```
too_many_positional_options_error()

class unknown_option:public hpx::program_options::error_with_no_option_name
    #include <errors.hpp> Class thrown when option name is not recognized.
```

Public Functions

```
unknown_option(const std::string &original_token = "")

~unknown_option()

class validation_error:public hpx::program_options::error_with_option_name
    #include <errors.hpp> Class thrown when value of option is incorrect.

Subclassed by hpx::program_options::invalid_bool_value, hpx::program_options::invalid_option_value
```

Public Types

```
enum kind_t
Values:
    multiple_values_not_allowed = 30
    at_least_one_value_required
    invalid_bool_value
    invalid_option_value
    invalid_option
```

Public Functions

```
validation_error (kind_t kind, const std::string &option_name = "", const std::string
&original_token = "", int option_style = 0)

~validation_error ()

kind_t kind() const
```

Protected Functions

```
std::string get_template (kind_t kind)
Used to convert kind_t to a related error text
```

Protected Attributes

```
kind_t m_kind

namespace hpx

namespace program_options
```

Typedefs

```
using option = basic_option<char>
using woption = basic_option<wchar_t>

template<class Char>
class basic_option
#include <option.hpp> Option found in input source. Contains a key and a value. The key, in turn,
can be a string (name of an option), or an integer (position in input source) – in case no name is
specified. The latter is only possible for command line. The template parameter specifies the type of
char used for storing the option's value.
```

Public Functions

```
basic_option()

basic_option (const std::string &xstring_key, const std::vector<std::string> &xvalue)
```

Public Members

```
std::string string_key
String key of this option. Intentionally independent of the template parameter.

int position_key
Position key of this option. All options without an explicit name are sequentially numbered
starting from 0. If an option has explicit name, ‘position_key’ is equal to -1. It is possible that
both position_key and string_key is specified, in case name is implicitly added.
```

```

std::vector<std::basic_string<Char>> value
    Option's value

std::vector<std::basic_string<Char>> original_tokens
    The original unchanged tokens this option was created from.

bool unregistered
    True if option was not recognized. In that case, ‘string_key’ and ‘value’ are results of purely syntactic parsing of source. The original tokens can be recovered from the “original_tokens” member.

bool case_insensitive
    True if string_key has to be handled case insensitive.

namespace hpx

namespace program_options

class duplicate_option_error : public hpx::program_options::error
#include <options_description.hpp> Class thrown when duplicate option description is found.

```

Public Functions

```

duplicate_option_error(const std::string &xwhat)

class option_description
#include <options_description.hpp> Describes one possible command line/config file option. There are two kinds of properties of an option. First describe it syntactically and are used only to validate input. Second affect interpretation of the option, for example default value for it or function that should be called when the value is finally known. Routines which perform parsing never use second kind of properties – they are side effect free.

See options_description

```

Public Types

```

enum match_result
    Values:
        no_match
        full_match
        approximate_match

```

Public Functions

```

option_description()

option_description(const char *name, const value_semantic *s)
    Initializes the object with the passed data.

```

Note: it would be nice to make the second parameter auto_ptr, to explicitly pass ownership. Unfortunately, it's often needed to create objects of types derived from ‘value_semantic’: *options_description d; d.add_options()("a", parameter<int>("n")->default_value(1));* Here, the static type returned by ‘parameter’ should be derived from *value_semantic*.

Alas, derived->base conversion for auto_ptr does not really work, see <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2000/n1232.pdf> http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#84

So, we have to use plain old pointers. Besides, users are not expected to use the constructor directly.

The ‘name’ parameter is interpreted by the following rules:

- if there’s no “,” character in ‘name’, it specifies long name
- otherwise, the part before “,” specifies long name and the part after – short name.

option_description (**const** char **name*, **const** *value_semantic* **s*, **const** char **description*)

Initializes the class with the passed data.

virtual ~option_description()

match_result match (**const** std::string &*option*, **bool** *approx*, **bool** *long_ignore_case*, **bool** *short_ignore_case*) **const**

Given ‘option’, specified in the input source, returns ‘true’ if ‘option’ specifies *this.

const std::string &key (**const** std::string &*option*) **const**

Returns the key that should identify the option, in particular in the *variables_map* class. The ‘option’ parameter is the option spelling from the input source. If option name contains ‘*’, returns ‘option’. If long name was specified, it’s the long name, otherwise it’s a short name with pre-pended ‘-’.

std::string canonical_display_name (**int** *canonical_option_style* = 0) **const**

Returns the canonical name for the option description to enable the user to recognize a matching option. 1) For short options ('-', '/'), returns the short name prefixed. 2) For long options (' / '-') returns the first long name prefixed 3) All other cases, returns the first long name (if present) or the short name, un-prefixed.

const std::string &long_name() **const**

const std::pair<const std::string*, std::size_t> long_names() **const**

const std::string &description() **const**

Explanation of this option.

std::shared_ptr<const value_semantic> semantic() **const**

Semantic of option’s value.

std::string format_name() **const**

Returns the option name, formatted suitably for usage message.

std::string format_parameter() **const**

Returns the parameter name and properties, formatted suitably for usage message.

Private Functions

`option_description &set_names (const char *name)`

Private Members

`std::string m_short_name`

a one-character “switch” name - with its prefix, so that this is either empty or has length 2 (e.g. “-c”)

`std::vector<std::string> m_long_names`

one or more names by which this option may be specified on a command-line or in a config file, which are not a single-letter switch. The names here are *without* any prefix.

`std::string m_description`

`std::shared_ptr<const value_semantic> m_value_semantic`

`class options_description`

`#include <options_description.hpp>` A set of option descriptions. This provides convenient interface for adding new option (the add_options) method, and facilities to search for options by name.

See here for option adding interface discussion.

See `option_description`

Public Functions

`options_description (unsigned line_length = m_default_line_length, unsigned min_description_length = m_default_line_length / 2)`

Creates the instance.

`options_description (const std::string &caption, unsigned line_length = m_default_line_length, unsigned min_description_length = m_default_line_length / 2)`

Creates the instance. The ‘caption’ parameter gives the name of this ‘`options_description`’ instance. Primarily useful for output. The ‘description_length’ specifies the number of columns that should be reserved for the description text; if the option text encroaches into this, then the description will start on the next line.

`void add (std::shared_ptr<option_description> desc)`

Adds new variable description. Throws duplicate_variable_error if either short or long name matches that of already present one.

`options_description &add (const options_description &desc)`

Adds a group of option description. This has the same effect as adding all `option_descriptions` in ‘`desc`’ individually, except that output operator will show a separate group. Returns `*this`.

`std::size_t get_option_column_width () const`

Find the maximum width of the option column, including options in groups.

`options_description_easy_init add_options ()`

Returns an object of implementation-defined type suitable for adding options to `options_description`. The returned object will have overloaded operator() with parameter type matching ‘`option_description`’ constructors. Calling the operator will create new `option_description` instance and add it.

```
const option_description &find(const std::string &name, bool approx, bool
                                long_ignore_case = false, bool short_ignore_case =
                                false) const
const option_description *find_nothrow(const std::string &name, bool approx,
                                       bool long_ignore_case = false, bool
                                       short_ignore_case = false) const
const std::vector<std::shared_ptr<option_description>> &options() const
void print(std::ostream &os, std::size_t width = 0) const
    Outputs 'desc' to the specified stream, calling 'f' to output each option_description element.
```

Public Static Attributes

```
const unsigned m_default_line_length
```

Private Types

```
using name2index_iterator = std::map<std::string, int>::const_iterator
using approximation_range = std::pair<name2index_iterator, name2index_iterator>
```

Private Members

```
std::string m_caption
const std::size_t m_line_length
const std::size_t m_min_description_length
std::vector<std::shared_ptr<option_description>> m_options
std::vector<char> belong_to_group
std::vector<std::shared_ptr<options_description>> groups
```

Friends

```
std::ostream &operator<< (std::ostream &os, const options_description &desc)
    Produces a human readable output of 'desc', listing options, their descriptions and allowed parameters. Other options_description instances previously passed to add will be output separately.

class options_description_easy_init
    #include <options_description.hpp> Class which provides convenient creation syntax to option_description.
```

Public Functions

```
options_description_easy_init(options_description *owner)
options_description_easy_init &operator()(const char *name, const char *description)
options_description_easy_init &operator()(const char *name, const value_semantic
                                         *s)
options_description_easy_init &operator()(const char *name, const value_semantic *s,
                                         const char *description)
```

Private Members

```
options_description *owner
namespace hpx
```

```
namespace program_options
```

TypeDefs

```
using parsed_options = basic_parsed_options<char>
using wparsed_options = basic_parsed_options<wchar_t>
using ext_parser = std::function<std::pair<std::string, std::string> (const std::string&) >
    Augments basic_parsed_options<wchar_t> with conversion from 'parsed_options'
using command_line_parser = basic_command_line_parser<char>
using wcommand_line_parser = basic_command_line_parser<wchar_t>
```

Enums

```
enum collect_unrecognized_mode
```

Controls if the ‘collect_unregistered’ function should include positional options, or not.

Values:

```
include_positional
exclude_positional
```

Functions

```
template<class Char>
basic_parsed_options<Char> parse_command_line(int argc, const Char *const argv[],
                                                const options_description&, int style
                                                = 0, std::function<std::pair<std::string,
                                                std::string> > const std::string&
                                                > ext = ext_parser()Creates instance of 'command_line_parser', passes parameters to it, and returns
                                                the result of calling the 'run' method.
```

```
template<class Char>
```

```
basic_parsed_options<Char> parse_config_file (std::basic_istream<Char>&, const  
options_description&, bool allow_unregistered = false)
```

Parse a config file.

Read from given stream.

```
template<class Char = char>  
basic_parsed_options<Char> parse_config_file (const char *filename, const  
options_description&, bool allow_unregistered = false)
```

Parse a config file.

Read from file with the given name. The character type is passed to the file stream.

```
template<class Char>  
std::vector<std::basic_string<Char>> collect_unrecognized (const  
std::vector<basic_option<Char>>&options, enum collect_unrecognized_mode  
mode)
```

Collects the original tokens for all named options with ‘unregistered’ flag set. If ‘mode’ is ‘include_positional’ also collects all positional options. Returns the vector of original tokens for all collected options.

```
parsed_options parse_environment (const options_description&, const  
std::function<std::string> std::string  
> &name_mapper) Parse environment.
```

For each environment variable, the ‘name_mapper’ function is called to obtain the option name. If it returns empty string, the variable is ignored.

This is done since naming of environment variables is typically different from the naming of command line options.

```
parsed_options parse_environment (const options_description&, const std::string &prefix)
```

Parse environment.

Takes all environment variables which start with ‘prefix’. The option name is obtained from variable name by removing the prefix and converting the remaining string into lower case.

```
parsed_options parse_environment (const options_description&, const char *prefix)
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This function exists to resolve ambiguity between the two above functions when second argument is of ‘char*’ type. There’s implicit conversion to both std::function and string.

```
std::vector<std::string> split_unix (const std::string &cmdline, const std::string &separator  
= "\t", const std::string &quote = "\"", const std::string  
&escape = "\\")
```

Splits a given string to a collection of single strings which can be passed to command_line_parser. The second parameter is used to specify a collection of possible separator chars used for splitting. The separator is defaulted to space “”. Splitting is done in a unix style way, with respect to quotes “” and escape characters “”

```
std::vector<std::wstring> hpx::program_options::split_unix (const std::wstring & cmd
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
template<class Char>
class basic_command_line_parser : private cmdline
#include <parsers.hpp> Command line parser.
```

The class allows one to specify all the information needed for parsing and to parse the command line. It is primarily needed to emulate named function parameters – a regular function with 5 parameters will be hard to use and creating overloads with a smaller number of parameters will be confusing.

For the most common case, the function `parse_command_line` is a better alternative.

There are two typedefs – `command_line_parser` and `wcommand_line_parser`, for `charT == char` and `charT == wchar_t` cases.

Public Functions

```
basic_command_line_parser(const std::vector<std::basic_string<Char>> &args)
    Creates a command line parser for the specified arguments list. The ‘args’ parameter should not include program name.

basic_command_line_parser(int argc, const Char *const argv[])
    Creates a command line parser for the specified arguments list. The parameters should be the same as passed to ‘main’.

basic_command_line_parser &options(const options_description &desc)
    Sets options descriptions to use.

basic_command_line_parser &positional(const positional_options_description &desc)
    Sets positional options description to use.

basic_command_line_parser &style(int)
    Sets the command line style.

basic_command_line_parser &extra_parser(ext_parser)
    Sets the extra parsers.

basic_parsed_options<Char> run()
    Parses the options and returns the result of parsing. Throws on error.

basic_command_line_parser &allow_unregistered()
    Specifies that unregistered options are allowed and should be passed though. For each command like token that looks like an option but does not contain a recognized name, an instance of basic_option<charT> will be added to result, with ‘unrecognized’ field set to ‘true’. It’s possible to collect all unrecognized options with the ‘collect_unrecognized’ function.

basic_command_line_parser &extra_style_parser(style_parser s)
```

Private Members

```
const options_description *m_desc
```

```
template<class Char>
class basic_parsed_options
#include <parsers.hpp> Results of parsing an input source. The primary use of this class is passing information from parsers component to value storage component. This class does not makes much sense itself.
```

Public Functions

```
basic_parsed_options(const options_description *xdescription, int options_prefix = 0)
```

Public Members

std::vector<basic_option<Char>> options

Options found in the source.

const options_description *description

Options description that was used for parsing. Parsers should return pointer to the instance of *option_description* passed to them, and issues of lifetime are up to the caller. Can be NULL.

int m_options_prefix

Mainly used for the diagnostic messages in exceptions. The canonical option prefix for the parser which generated these results, depending on the settings for *basic_command_line_parser::style()* or *cmdline::style()*. In order of precedence of command_line_style enums: allow_long allow_long_disguise allow_dash_for_short allow_slash_for_short

template<>

class basic_parsed_options<wchar_t>

#include <parsers.hpp> Specialization of *basic_parsed_options* which:

- provides convenient conversion from *basic_parsed_options<char>*
- stores the passed char-based options for later use.

Public Functions

```
basic_parsed_options(const basic_parsed_options<char> &po)
```

Constructs wrapped options from options in UTF8 encoding.

Public Members

std::vector<basic_option<wchar_t>> options

const options_description *description

basic_parsed_options<char> utf8_encoded_options

Stores UTF8 encoded options that were passed to constructor, to avoid reverse conversion in some cases.

int m_options_prefix

Mainly used for the diagnostic messages in exceptions. The canonical option prefix for the parser which generated these results, depending on the settings for *basic_command_line_parser::style()* or *cmdline::style()*. In order of precedence of command_line_style enums: allow_long allow_long_disguise allow_dash_for_short allow_slash_for_short

namespace hpx

namespace program_options

```
class positional_options_description
#include <positional_options.hpp> Describes positional options.
```

The class allows to guess option names for positional options, which are specified on the command line and are identified by the position. The class uses the information provided by the user to associate a name with every positional option, or tell that no name is known.

The primary assumption is that only the relative order of the positional options themselves matters, and that any interleaving ordinary options don't affect interpretation of positional options.

The user initializes the class by specifying that first N positional options should be given the name X1, following M options should be given the name X2 and so on.

Public Functions

```
positional_options_description()
```

```
positional_options_description &add(const char *name, int max_count)
```

Species that up to ‘max_count’ next positional options should be given the ‘name’. The value of ‘-1’ means ‘unlimited’. No calls to ‘add’ can be made after call with ‘max_value’ equal to ‘-1’.

```
unsigned max_total_count() const
```

Returns the maximum number of positional options that can be present. Can return (numeric_limits<unsigned>::max)() to indicate unlimited number.

```
const std::string &name_for_position(unsigned position) const
```

Returns the name that should be associated with positional options at ‘position’. Precondition: position < max_total_count()

Private Members

```
std::vector<std::string> m_names
```

```
std::string m_trailing
```

```
namespace hpx
```

```
namespace program_options
```

Functions

```
template<class T>
```

```
typed_value<T> *value()
```

Creates a typed_value<T> instance. This function is the primary method to create *value_semantic* instance for a specific type, which can later be passed to ‘option_description’ constructor. The second overload is used when it’s additionally desired to store the value of option into program variable.

```
template<class T>
```

```
typed_value<T> *value(T *v)
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
template<class T>
```

```
typed_value<T, wchar_t> *wvalue()
```

Creates a typed_value<T> instance. This function is the primary method to create *value_semantic* instance for a specific type, which can later be passed to ‘option_description’ constructor.

```
template<class T>
```

```
typed_value<T, wchar_t> *wvalue(T *v)
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
typed_value<bool> *bool_switch()
```

Works the same way as the ‘value<bool>’ function, but the created *value_semantic* won’t accept any explicit value. So, if the option is present on the command line, the value will be ‘true’.

```
typed_value<bool> *bool_switch(bool *v)
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
template<class T, class Char = char>
```

```
class typed_value : public hpx::program_options::value_semantic_codecvt_helper<char>, public hpx::program
```

#include <value_semantic.hpp> Class which handles value of a specific type.

Public Functions

```
typed_value(T *store_to)
```

Ctor. The ‘store_to’ parameter tells where to store the value when it’s known. The parameter can be NULL.

```
typed_value *default_value(const T &v)
```

Specifies default value, which will be used if none is explicitly specified. The type ‘T’ should provide operator<< for ostream.

```
typed_value *default_value(const T &v, const std::string &textual)
```

Specifies default value, which will be used if none is explicitly specified. Unlike the above overload, the type ‘T’ need not provide operator<< for ostream, but textual representation of default value must be provided by the user.

```
typed_value *implicit_value(const T &v)
```

Specifies an implicit value, which will be used if the option is given, but without an adjacent value. Using this implies that an explicit value is optional,

```
typed_value *value_name(const std::string &name)
```

Specifies the name used to to the value in help message.

```
typed_value *implicit_value(const T &v, const std::string &textual)
```

Specifies an implicit value, which will be used if the option is given, but without an adjacent value. Using this implies that an explicit value is optional, but if given, must be strictly adjacent to the option, i.e.: ‘-ovalue’ or ‘option=value’. Giving ‘-o’ or ‘option’ will cause the implicit value to be applied. Unlike the above overload, the type ‘T’ need not provide operator<< for ostream, but textual representation of default value must be provided by the user.

```
typed_value *notifier(std::function<void> f)
```

>fSpecifies a function to be called when the final value is determined.

```
typed_value *composing()
```

Specifies that the value is composing. See the ‘is_composing’ method for explanation.

`typed_value *multitoken()`
 Specifies that the value can span multiple tokens.

`typed_value *zero_tokens()`
 Specifies that no tokens may be provided as the value of this option, which means that only presence of the option is significant. For such option to be useful, either the ‘validate’ function should be specialized, or the ‘implicit_value’ method should be also used. In most cases, you can use the ‘bool_switch’ function instead of using this method.

`typed_value *required()`
 Specifies that the value must occur.

`std::string name() const`
 Returns the name of the option. The name is only meaningful for automatic help message.

`bool is_composing() const`
 Returns true if values from different sources should be composed. Otherwise, value from the first source is used and values from other sources are discarded.

`unsigned min_tokens() const`
 The minimum number of tokens for this option that should be present on the command line.

`unsigned max_tokens() const`
 The maximum number of tokens for this option that should be present on the command line.

`bool is_required() const`
 Returns true if value must be given. Non-optional value

`void xparse(hpx::any_nonser &value_store, const std::vector<std::basic_string<Char>> &new_tokens) const`
 Creates an instance of the ‘validator’ class and calls its operator() to perform the actual conversion.

`virtual bool apply_default(hpx::any_nonser &value_store) const`
 If default value was specified via previous call to ‘default_value’, stores that value into ‘value_store’. Returns true if default value was stored.

`void notify(const hpx::any_nonser &value_store) const`
 If an address of variable to store value was specified when creating *this, stores the value there. Otherwise, does nothing.

`const std::type_info &value_type() const`

Private Members

```
T *m_store_to
std::string m_value_name
hpx::any_nonser m_default_value
std::string m_default_value_as_text
hpx::any_nonser m_implicit_value
std::string m_implicit_value_as_text
bool m_composing
bool m_implicit
```

```
bool m_multitoken  
bool m_zero_tokens  
bool m_required  
std::function<void (const T&) > m_notifier  
  
class typed_value_base  
#include <value_semantic.hpp> Base class for all option that have a fixed type, and are willing to  
announce this type to the outside world. Any ‘value_semantics’ for which you want to find out the  
type can be dynamic_cast-ed to typed_value_base. If conversion succeeds, the ‘type’ method can be  
called.
```

Subclassed by *hpx::program_options::typed_value< T, Char >*

Public Functions

```
virtual const std::type_info &value_type() const = 0  
  
virtual ~typed_value_base()  
  
class untyped_value : public hpx::program_options::value_semantic_codecvt_helper<char>  
#include <value_semantic.hpp> Class which specifies a simple handling of a value: the value will  
have string type and only one token is allowed.
```

Public Functions

```
untyped_value(bool zero_tokens = false)  
  
std::string name() const  
    Returns the name of the option. The name is only meaningful for automatic help message.  
  
unsigned min_tokens() const  
    The minimum number of tokens for this option that should be present on the command line.  
  
unsigned max_tokens() const  
    The maximum number of tokens for this option that should be present on the command line.  
  
bool is_composing() const  
    Returns true if values from different sources should be composed. Otherwise, value from the first  
    source is used and values from other sources are discarded.  
  
bool is_required() const  
    Returns true if value must be given. Non-optional value  
  
void xparse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens)  
    const  
    If ‘value_store’ is already initialized, or new_tokens has more than one elements, throws. Other-  
    wise, assigns the first string from ‘new_tokens’ to ‘value_store’, without any modifications.  
  
bool apply_default(hpx::any_nonser&) const  
    Does nothing.  
  
void notify(const hpx::any_nonser&) const  
    Does nothing.
```

Private Members

```
bool m_zero_tokens

class value_semantic
#include <value_semantic.hpp> Class which specifies how the option's value is to be parsed and converted into C++ types.

Subclassed by hpx::program_options::value_semantic_codecvt_helper< char >, hpx::program_options::value_semantic_codecvt_helper< wchar_t >
```

Public Functions

virtual std::string name() const = 0

Returns the name of the option. The name is only meaningful for automatic help message.

virtual unsigned min_tokens() const = 0

The minimum number of tokens for this option that should be present on the command line.

virtual unsigned max_tokens() const = 0

The maximum number of tokens for this option that should be present on the command line.

virtual bool is_composing() const = 0

Returns true if values from different sources should be composed. Otherwise, value from the first source is used and values from other sources are discarded.

virtual bool is_required() const = 0

Returns true if value must be given. Non-optional value

virtual void parse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens, bool utf8) const = 0

Parses a group of tokens that specify a value of option. Stores the result in ‘value_store’, using whatever representation is desired. May be called several times if value of the same option is specified more than once.

virtual bool apply_default(hpx::any_nonser &value_store) const = 0

Called to assign default value to ‘value_store’. Returns true if default value is assigned, and false if no default value exists.

virtual void notify(const hpx::any_nonser &value_store) const = 0

Called when final value of an option is determined.

virtual ~value_semantic()

template<class Char>

class value_semantic_codecvt_helper

#include <value_semantic.hpp> Helper class which perform necessary character conversions in the ‘parse’ method and forwards the data further.

template<>

class value_semantic_codecvt_helper<char> : public hpx::program_options::value_semantic

#include <value_semantic.hpp> Helper conversion class for values that accept ascii strings as input. Overrides the ‘parse’ method and defines new ‘xparse’ method taking std::string. Depending on whether input to parse is ascii or UTF8, will pass it to xparse unmodified, or with UTF8->ascii conversion.

Subclassed by hpx::program_options::typed_value< T, Char >, hpx::program_options::untyped_value

Protected Functions

```
virtual void xparse(hpx::any_nonser &value_store, const std::vector<std::string>&new_tokens) const = 0
```

Private Functions

```
void parse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens, bool utf8) const
```

Parses a group of tokens that specify a value of option. Stores the result in ‘value_store’, using whatever representation is desired. May be called several times if value of the same option is specified more than once.

```
template<>
class value_semantic_codecvt_helper<wchar_t> : public hpx::program_options::value_semantic
#include <value_semantic.hpp> Helper conversion class for values that accept ascii strings as input. Overrides the ‘parse’ method and defines new ‘xparse’ method taking std::wstring. Depending on whether input to parse is ascii or UTF8, will recode input to Unicode, or pass it unmodified.
```

Protected Functions

```
virtual void xparse(hpx::any_nonser &value_store, const std::vector<std::wstring>&new_tokens) const = 0
```

Private Functions

```
void parse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens, bool utf8) const
```

Parses a group of tokens that specify a value of option. Stores the result in ‘value_store’, using whatever representation is desired. May be called several times if value of the same option is specified more than once.

```
namespace hpx
```

```
namespace program_options
```

Functions

```
void store(const basic_parsed_options<char> &options, variables_map &m, bool utf8 = false)
```

Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an option, that value is not changed, even if ‘options’ specify some value.

```
void store(const basic_parsed_options<wchar_t> &options, variables_map &m)
```

Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an option, that value is not changed, even if ‘options’ specify some value. This is wide character variant.

```
void notify(variables_map &m)
```

Runs all ‘notify’ function for options in ‘m’.

```
class abstract_variables_map
#include <variables_map.hpp> Implements string->string mapping with convenient value casting facilities.

Subclassed by hpx::program_options::variables_map
```

Public Functions

```
abstract_variables_map()
abstract_variables_map(const abstract_variables_map *next)
virtual ~abstract_variables_map()
const variable_value &operator[](const std::string &name) const
```

Obtains the value of variable ‘name’, from *this and possibly from the chain of variable maps.

- if there’s no value in *this.
 - if there’s next variable map, returns value from it
 - otherwise, returns empty value
- if there’s defaulted value
 - if there’s next variable map, which has a non-defaulted value, return that
 - otherwise, return value from *this
- if there’s a non-defaulted value, returns it.

```
void next(abstract_variables_map *next)
```

Sets next variable map, which will be used to find variables not found in *this.

Private Functions

```
virtual const variable_value &get(const std::string &name) const = 0
```

Returns value of variable ‘name’ stored in *this, or empty value otherwise.

Private Members

```
const abstract_variables_map *m_next
```

```
class variable_value
```

```
#include <variables_map.hpp> Class holding value of option. Contains details about how the value is set and allows to conveniently obtain the value.
```

Public Functions

```
variable_value()
```

```
variable_value(const hpx::any_nonser &xv, bool xdefaulted)
```

```
template<class T>
```

```
const T &as() const
```

If stored value is of type T, returns that value. Otherwise, throws boost::bad_any_cast exception.

```
template<class T>
```

T &**as** ()

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool empty () const

Returns true if no value is stored.

bool defaulted () const

Returns true if the value was not explicitly given, but has default value.

hpx::any_nonser &value () const

Returns the contained value.

hpx::any_nonser &value ()

Returns the contained value.

Private Members

hpx::any_nonser v

bool m_defaulted

std::shared_ptr<const value_semantic> m_value_semantic

Friends

friend hpx::program_options::variables_map

void store (const basic_parsed_options<char> &options, variables_map &m, bool utf8)

Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an option, that value is not changed, even if ‘options’ specify some value.

class variables_map : public hpx::program_options::abstract_variables_map, public std::map<std::string, variable_value> Concrete variables map which store variables in real map.
#include <variables_map.hpp>

This class is derived from *std::map<std::string, variable_value>*, so you can use all map operators to examine its content.

Public Functions

variables_map ()

variables_map (const abstract_variables_map *next)

const variable_value &operator[] (const std::string &name) const

void clear ()

void notify ()

Private Functions

const variable_value &get (const std::string &name) const
 Implementation of abstract_variables_map::get which does ‘find’ in *this.

Private Members

std::set<std::string> m_final
 Names of option with ‘final’ values – which should not be changed by subsequence assignments.

std::map<std::string, std::string> m_required
 Names of required options, filled by parser which has access to *options_description*. The map values are the “canonical” names for each corresponding option. This is useful in creating diagnostic messages when the option is absent.

Friends

void store (const basic_parsed_options<char> &options, variables_map &xm, bool utf8)
 Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an option, that value is not changed, even if ‘options’ specify some value.

Defines

HPX_PROGRAM_OPTIONS_VERSION

The version of the source interface. The value will be incremented whenever a change is made which might cause compilation errors for existing code.

HPX_PROGRAM_OPTIONS_IMPLICIT_VALUE_NEXT_TOKEN

properties

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

namespace experimental

Variables

hpx::experimental::prefer_t prefer

struct prefer_t : public hpx::functional::detail::tag_fallback<prefer_t>

Friends

```
template<typename Tag, typename ...Tn>
friend constexpr auto tagFallbackInvoke(prefer_t, Tag const &tag, Tn&&...
                                         tn)

template<typename Tag, typename T0, typename ...Tn>
friend constexpr auto tagFallbackInvoke(prefer_t, Tag, T0 &&t0, Tn&&...)
```

resiliency

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

namespace resiliency

namespace experimental

Functions

```
template<typename Pred, typename F, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tagInvoke(async_replay_validator,
                                              std::size_t
                                              n,
                                              Pred
                                              &&pred,
                                              F
                                              &&f,
                                              Ts&&...
                                              ts)

template<typename F, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tagInvoke(async_replay_t,
                                              std::size_t
                                              n,
                                              F
                                              &&f,
                                              Ts&&...
                                              ts)
```

namespace hpx

namespace resiliency

namespace experimental

Functions

```
template<typename Executor, typename Pred, typename F, typename ...Ts>
decltype(auto) tag_invoke(async_replay_validate_t, Executor &&exec, std::size_t n, Pred
    &&pred, F &&f, Ts&&... ts)

template<typename Executor, typename F, typename ...Ts>
decltype(auto) tag_invoke(async_replay_t, Executor &&exec, std::size_t n, F &&f, Ts&&...
    ts)

namespace hpx
```

```
namespace resiliency
```

```
namespace experimental
```

Functions

```
template<typename Vote, typename Pred, typename F, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke(async_replicate_vote
    std::size_t
    n,
    Vote
    &&vote,
    Pred
    &&pred,
    F
    &&f,
    Ts&&...
    ts)
```

```
template<typename Vote, typename F, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke(async_replicate_vote
    std::size_t
    n,
    Vote
    &&vote,
    F
    &&f,
    Ts&&...
    ts)
```

```
template<typename Pred, typename F, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke(async_replicate_val
    std::size_t
    n,
    Pred
    &&pred,
    F
    &&f,
    Ts&&...
    ts)
```

```
template<typename F, typename ...Ts>
```

```
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke (async_replicate_t,
                                                                 std::size_t
                                                                 n,
                                                                 F
                                                                 &&f,
                                                                 Ts&&...
                                                                 ts)

namespace hpx

namespace resiliency

namespace experimental

Functions
```

```
template<typename Executor, typename Vote, typename Pred, typename F, typename ...Ts>
decltype(auto) tag_invoke (async_replicate_vote_validate_t, Executor &&exec, std::size_t n,
                           Vote &&vote, Pred &&pred, F &&f, Ts&&... ts)
```

```
template<typename Executor, typename Vote, typename F, typename ...Ts>
decltype(auto) tag_invoke (async_replicate_vote_t, Executor &&exec, std::size_t n, Vote
                           &&vote, F &&f, Ts&&... ts)
```

```
template<typename Executor, typename Pred, typename F, typename ...Ts>
decltype(auto) tag_invoke (async_replicate_validate_t, Executor &&exec, std::size_t n, Pred
                           &&pred, F &&f, Ts&&... ts)
```

```
template<typename Executor, typename F, typename ...Ts>
decltype(auto) tag_invoke (async_replicate_t, Executor &&exec, std::size_t n, F &&f,
                           Ts&&... ts)
```

```
namespace hpx
```

```
namespace resiliency
```

```
namespace experimental
```

```
Functions
```

```
template<typename BaseExecutor, typename Validate>
replay_executor<BaseExecutor, typename std::decay<Validate>::type> make_replay_executor (BaseExecutor
                                         &exec,
                                         std::size_t
                                         n,
                                         Val-
                                         i-
                                         date
                                         &&val-
                                         i-
                                         date)
```

```
template<typename BaseExecutor>
```

```
replay_executor<BaseExecutor, detail::replay_validator> make_replay_executor (BaseExecutor
&exec,
std::size_t
n)
```

```
template<typename BaseExecutor, typename Validate>
class replay_executor
```

Public Types

```
template<>
using execution_category = typename BaseExecutor::execution_category
template<>
using executor_parameters_type = typename BaseExecutor::executor_parameters_type
template<typename Result>
using future_type = typename hpx::parallel::execution::executor_future<BaseExecutor, Result>::type
```

Public Functions

```
template<typename F>
replay_executor (BaseExecutor &exec, std::size_t n, F &&f)
bool operator== (replay_executor const &rhs) const
bool operator!= (replay_executor const &rhs) const
replay_executor const &context () const
template<typename F, typename ...Ts>
decltype(auto) async_execute (F &&f, Ts&&... ts) const
template<typename F, typename S, typename ...Ts>
decltype(auto) bulk_async_execute (F &&f, S const &shape, Ts&&... ts) const
```

Public Static Attributes

```
constexpr int num_spread = 4
constexpr int num_tasks = 128
```

Private Members

```
BaseExecutor &exec_
std::size_t replay_count_
Validate validator_
```

```
namespace hpx
```

```
namespace resiliency
```

```
namespace experimental
```

Functions

```
template<typename BaseExecutor, typename Voter, typename Validate>
replicate_executor<BaseExecutor, typename std::decay<Voter>::type, typename std::decay<Validate>::type> make_
```

```
template<typename BaseExecutor, typename Validate>
replicate_executor<BaseExecutor, detail::replicate_voter, typename std::decay<Validate>::type> make_replicat
```

```
template<typename BaseExecutor>
replicate_executor<BaseExecutor, detail::replicate_voter, detail::replicate_validator> make_replicate_executor
```

```
template<typename BaseExecutor, typename Vote, typename Validate>
class replicate_executor
```

Public Types

```
template<>
using execution_category = typename BaseExecutor::execution_category
template<>
using executor_parameters_type = typename BaseExecutor::executor_parameters_type
template<typename Result>
using future_type = typename hpx::parallel::execution::executor_future<BaseExecutor, Result>::type
```

Public Functions

```
template<typename V, typename F>
replicate_executor(BaseExecutor &exec, std::size_t n, V &&v, F &&f)

bool operator==(replicate_executor const &rhs) const

bool operator!=(replicate_executor const &rhs) const

replicate_executor const &context() const

template<typename F, typename ...Ts>
decltype(auto) async_execute(F &&f, Ts&&... ts) const

template<typename F, typename S, typename ...Ts>
decltype(auto) bulk_async_execute(F &&f, S const &shape, Ts&&... ts) const
```

Public Static Attributes

```
constexpr int num_spread = 4
constexpr int num_tasks = 128
```

Private Members

```
BaseExecutor &exec_
std::size_t replicate_count_
Vote voter_
Validate validator_
```

namespace hpx

namespace resiliency

namespace experimental

Variables

```
hpx::resiliency::experimental::async_replay_validate_t async_replay_validate
hpx::resiliency::experimental::async_replay_t async_replay
hpx::resiliency::experimental::dataflow_replay_validate_t dataflow_replay_validate
hpx::resiliency::experimental::dataflow_replay_t dataflow_replay
hpx::resiliency::experimental::async_replicate_vote_validate_t async_replicate_vote_validate
hpx::resiliency::experimental::async_replicate_vote_t async_replicate_vote
hpx::resiliency::experimental::async_replicate_validate_t async_replicate_validate
hpx::resiliency::experimental::async_replicate_t async_replicate
hpx::resiliency::experimental::dataflow_replicate_vote_validate_t dataflow_replicate_vote_validate
```

```
hpx::resiliency::experimental::dataflow_replicate_vote_t dataflow_replicate_vote
hpx::resiliency::experimental::dataflow_replicate_validate_t dataflow_replicate_validate
hpx::resiliency::experimental::dataflow_replicate_t dataflow_replicate

struct async_replay_t : public hpx::functional::tag<async_replay_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching given function f repeatedly. Repeat launching on error exactly n times (except if abort_replay_exception is thrown).

struct async_replay_validate_t : public hpx::functional::tag<async_replay_validate_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f. repeatedly. Verify the result of those invocations using the given predicate pred. Repeat launching on error exactly n times (except if abort_replay_exception is thrown).

struct async_replicate_t : public hpx::functional::tag<async_replicate_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f exactly n times concurrently. Verify the result of those invocations by checking for exception. Return the first valid result.

struct async_replicate_validate_t : public hpx::functional::tag<async_replicate_validate_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f exactly n times concurrently. Verify the result of those invocations using the given predicate pred. Return the first valid result.

struct async_replicate_vote_t : public hpx::functional::tag<async_replicate_vote_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f exactly n times concurrently. Verify the result of those invocations using the given predicate pred. Run all the valid results against a user provided voting function. Return the valid output.

struct async_replicate_vote_validate_t : public hpx::functional::tag<async_replicate_vote_validate_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f exactly n times concurrently. Verify the result of those invocations using the given predicate pred. Run all the valid results against a user provided voting function. Return the valid output.

struct dataflow_replay_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replay_t, async_replay_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f. repeatedly. Repeat launching on error exactly n times.

    Delay the invocation of f if any of the arguments to f are futures.

struct dataflow_replay_validate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replay_validate_t, async_replay_validate_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f. repeatedly. Verify the result of those invocations using the given predicate pred. Repeat launching on error exactly n times.

    Delay the invocation of f if any of the arguments to f are futures.

struct dataflow_replicate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_t, async_replicate_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f exactly n times concurrently. Return the first valid result.

    Delay the invocation of f if any of the arguments to f are futures.

struct dataflow_replicate_validate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_validate_t, async_replicate_validate_t>
    #include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function f exactly n times concurrently. Verify the result of those invocations using the given predicate pred. Return the first valid result.
```

function f exactly n times concurrently. Verify the result of those invocations using the given predicate $pred$. Return the first valid result.

Delay the invocation of f if any of the arguments to f are futures.

```
struct dataflow_replicate_vote_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicat
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given
function  $f$  exactly  $n$  times concurrently. Run all the valid results against a user provided vot-
ing function. Return the valid output.
```

Delay the invocation of f if any of the arguments to f are futures.

```
struct dataflow_replicate_vote_validate_t : public hpx::resiliency::experimental::tag_deferred<d
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given
function  $f$  exactly  $n$  times concurrently. Run all the valid results against a user provided vot-
ing function. Return the valid output.
```

Delay the invocation of f if any of the arguments to f are futures.

```
template<typename Tag, typename BaseTag>
struct tag_deferred : public hpx::functional::tag<Tag>
```

Friends

```
template<typename ...Args>
auto tag_invoke (Tag, Args&&... args)
```

Defines

```
HPX_RESILIENCY_VERSION_FULL
HPX_RESILIENCY_VERSION_MAJOR
HPX_RESILIENCY_VERSION_MINOR
HPX_RESILIENCY_VERSION_SUBMINOR
HPX_RESILIENCY_VERSION_DATE
namespace hpx
```

```
namespace resiliency
```

```
namespace experimental
```

Functions

```
unsigned int major_version ()
unsigned int minor_version ()
unsigned int subminor_version ()
unsigned long full_version ()
std::string full_version_str()
```

resource_partitioner

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx**namespace resource****class core****Public Functions**

```
core (std::size_t id = invalid_core_id, numa_domain *domain = nullptr)  
std::vector<pu> const &pus () const  
std::size_t id () const
```

Private Functions

```
std::vector<core> cores_sharing_numa_domain ()
```

Private Members

```
std::size_t id_  
numa_domain *domain_  
std::vector<pu> pus_
```

Private Static Attributes

```
constexpr const std::size_t invalid_core_id = std::size_t(-1)
```

Friends

```
friend hpx::resource::pu  
friend hpx::resource::numa_domain  
class numa_domain
```

Public Functions

```
numa_domain (std::size_t id = invalid_numa_domain_id)
    std::vector<core> const &cores () const
    std::size_t id () const
```

Private Members

```
std::size_t id_
std::vector<core> cores_
```

Private Static Attributes

```
constexpr const std::size_t invalid_numa_domain_id = std::size_t(-1)
```

Friends

```
friend hpx::resource::pu
friend hpx::resource::core

class partitioner
```

Public Functions

```
void create_thread_pool (std::string const &name, scheduling_policy sched = scheduling_policy::unspecified,
                        hpx::threads::policies::scheduler_mode mode = hpx::threads::policies::scheduler_mode::default_mode)

void create_thread_pool (std::string const &name, scheduler_function scheduler_creation)

void set_default_pool_name (std::string const &name)
const std::string &get_default_pool_name () const

void add_resource (hpx::resource::pu const &p, std::string const &pool_name,
                   std::size_t num_threads = 1)

void add_resource (hpx::resource::pu const &p, std::string const &pool_name, bool exclusive, std::size_t num_threads = 1)

void add_resource (std::vector<hpx::resource::pu> const &pv, std::string const &pool_name, bool exclusive = true)

void add_resource (hpx::resource::core const &c, std::string const &pool_name, bool exclusive = true)

void add_resource (std::vector<hpx::resource::core> &cv, std::string const &pool_name, bool exclusive = true)

void add_resource (hpx::resource::numa_domain const &nd, std::string const &pool_name, bool exclusive = true)
```

```
void add_resource (std::vector<hpx::resource::numa_domain> const &nadv, std::string  
                    const &pool_name, bool exclusive = true)  
  
std::vector<numa_domain> const &numa_domains () const  
  
std::size_t get_number_requested_threads ()  
  
hpx::threads::topology const &get_topology () const  
  
void configure_pools ()
```

Private Functions

```
partitioner (resource::partitioner_mode rpmode, hpx::util::section rtcfg,  
                hpx::threads::policies::detail::affinity_data affinity_data)
```

Private Members

```
detail::partitioner &partitioner_  
  
class pu
```

Public Functions

```
pu (std::size_t id = invalid_pu_id, core *core = nullptr, std::size_t thread_occupancy = 0)  
  
std::size_t id () const
```

Private Functions

```
std::vector<pu> pus_sharing_core ()  
  
std::vector<pu> pus_sharing_numa_domain ()
```

Private Members

```
std::size_t id_  
core *core_  
std::size_t thread_occupancy_  
std::size_t thread_occupancy_count_
```

Private Static Attributes

```
constexpr const std::size_t invalid_pu_id = std::size_t(-1)
```

Friends

```
friend hpx::resource::core
friend hpx::resource::numa_domain
```

```
namespace hpx
```

```
namespace resource
```

TypeDefs

```
using scheduler_function = util::function_nonser<std::unique_ptr<hpx::threads::thread_pool_base> (hpx::thread
hpx::thread
```

Enums

enum partitioner_mode

This enumeration describes the modes available when creating a resource partitioner.

Values:

mode_default = 0

Default mode.

mode_allow_oversubscription = 1

Allow processing units to be oversubscribed, i.e. multiple worker threads to share a single processing unit.

mode_allow_dynamic_pools = 2

Allow worker threads to be added and removed from thread pools.

enum scheduling_policy

This enumeration lists the available scheduling policies (or schedulers) when creating thread pools.

Values:

user_defined = -2

unspecified = -1

local = 0

local_priority_fifo = 1

local_priority_lifo = 2

static_ = 3

static_priority = 4

abp_priority_fifo = 5

abp_priority_lifo = 6

shared_priority = 7

Functions

```
detail::partitioner &get_partitioner()
```

May be used anywhere in code and returns a reference to the single, global resource partitioner.

```
bool is_partitioner_valid()
```

Returns true if the resource partitioner has been initialized. Returns false otherwise.

runtime_configuration

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
namespace hpx
```

```
namespace agas
```

Enums

```
enum service_mode
```

Values:

```
service_mode_invalid = -1
```

```
service_mode_bootstrap = 0
```

```
service_mode_hosted = 1
```

Defines

```
HPX_REGISTER_COMMANDLINE_REGISTRY (RegistryType, componentname)
```

The macro *HPX_REGISTER_COMMANDLINE_REGISTRY* is used to register the given component factory with *Hpx.Plugin*. This macro has to be used for each of the components.

```
HPX_REGISTER_COMMANDLINE_REGISTRY_DYNAMIC (RegistryType, componentname)
```

```
HPX_REGISTER_COMMANDLINE_OPTIONS ()
```

The macro *HPX_REGISTER_COMMANDLINE_OPTIONS* is used to define the required *Hpx.Plugin* entry point for the command line option registry. This macro has to be used in not more than one compilation unit of a component module.

```
HPX_REGISTER_COMMANDLINE_OPTIONS_DYNAMIC ()
```

```
namespace hpx
```

```
namespace components
```

```
struct component_commandline_base
```

#include <component_commandline_base.hpp> The *component_commandline_base* has to be used as a base class for all component command-line line handling registries.

Public Functions

```
virtual ~component_commandline_base()
virtual hpx::program_options::options_description add_commandline_options() = 0
    Return any additional command line options valid for this component.
```

Return The module is expected to fill a options_description object with any additional command line options this component will handle.

Note This function will be executed by the runtime system during system startup.

Defines

HPX_REGISTER_COMPONENT_FACTORY (*componentname*)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the component factories.

HPX_REGISTER_COMPONENT_MODULE ()

This macro is used to define the required Hpx.Plugin entry points. This macro has to be used in exactly one compilation unit of a component module.

HPX_REGISTER_COMPONENT_MODULE_DYNAMIC ()

Defines

HPX_REGISTER_COMPONENT_REGISTRY (*RegistryType, componentname*)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the components.

HPX_REGISTER_COMPONENT_REGISTRY_DYNAMIC (*RegistryType, componentname*)

HPX_REGISTER_REGISTRY_MODULE ()

This macro is used to define the required Hpx.Plugin entry points. This macro has to be used in exactly one compilation unit of a component module.

HPX_REGISTER_REGISTRY_MODULE_DYNAMIC ()

```
namespace hpx
```

```
namespace components
```

struct component_registry_base

```
#include <component_registry_base.hpp>
```

The component_registry_base has to be used as a base class for all component registries.

Public Functions

```
virtual ~component_registry_base()

virtual bool get_component_info (std::vector<std::string> &fillini, std::string const
&filepath, bool is_static = false) = 0
```

Return the ini-information for all contained components.

Return Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

Parameters

- *fillini*: [in, out] The module is expected to fill this vector with the ini-information (one line per vector element) for all components implemented in this module.

```
virtual void register_component_type () = 0
```

Return the unique identifier of the component type this factory is responsible for.

Return Returns the unique identifier of the component type this factory instance is responsible for. This function throws on any error.

Parameters

- *locality*: [in] The id of the locality this factory is responsible for.
- *agas_client*: [in] The AGAS client to use for component id registration (if needed).

```
namespace hpx
```

```
namespace util
```

Functions

```
bool handle_ini_file (section &ini, std::string const &loc)
```

```
bool handle_ini_file_env (section &ini, char const *env_var, char const *file_suffix =
nullptr)
```

```
bool init_ini_data_base (section &ini, std::string &hpx_ini_file)
```

```
std::vector<std::shared_ptr<components::component_registry_base>> load_component_factory_static (util::sec
&ini,
std::string name,
hpx::util::get_fact
er->ror_code
&ec
=
throws)
```

```
void merge_component_inis (section &ini)
```

```
std::vector<std::shared_ptr<plugins::plugin_registry_base>> init_ini_data_default (std::string
    const
    &libs,
    sec-
    tion
    &ini,
    std::map<std::string,
    filesys-
    tem::path>
    &base-
    names,
    std::map<std::string,
    hpx::util::plugin::dll>
    &mod-
    ules,
    std::vector<std::shared_ptr<com-
    ponent_registry>>
    &com-
    ponent_registries)
```

Defines

`HPX_REGISTER_PLUGIN_BASE_REGISTRY` (*PluginType, name*)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the components.

`HPX_REGISTER_PLUGIN_REGISTRY_MODULE` ()

This macro is used to define the required Hpx.Plugin entry points. This macro has to be used in exactly one compilation unit of a component module.

`HPX_REGISTER_PLUGIN_REGISTRY_MODULE_DYNAMIC` ()

```
namespace hpx
```

```
namespace plugins
```

`struct plugin_registry_base`

#include <plugin_registry_base.hpp> The plugin_registry_base has to be used as a base class for all plugin registries.

Public Functions

`virtual ~plugin_registry_base` ()

`virtual bool get_plugin_info (std::vector<std::string> &fillini) = 0`

Return the configuration information for any plugin implemented by this module

Return Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

Parameters

- *fillini*: [in, out] The module is expected to fill this vector with the ini-information (one line per vector element) for all plugins implemented in this module.

```
virtual void init (int*, char***, util::runtime_configuration&)
namespace hpx
```

```
namespace util
```

```
class runtime_configuration : public section
```

Public Functions

```
runtime_configuration (char const *argv0, runtime_mode mode,
                      std::vector<std::string> const &extra_static_ini_defs =
                      {})

void reconfigure (std::string const &ini_file)

void reconfigure (std::vector<std::string> const &ini_defs)

std::vector<std::shared_ptr<plugins::plugin_registry_base>> load_modules (std::vector<std::shared_ptr<component_registry>> &component_registries)

void load_components_static (std::vector<components::static_factory_load_data_type>
                            const &static_modules)

agas::service_mode get_agas_service_mode () const

std::uint32_t get_num_localities () const

void set_num_localities (std::uint32_t)

bool enable_networking () const

std::uint32_t get_first_used_core () const

void set_first_used_core (std::uint32_t)

std::size_t get_ipc_data_buffer_cache_size () const

std::size_t get_agas_local_cache_size (std::size_t dflt = HPX_AGAS_LOCAL_CACHE_SIZE)
                                         const

bool get_agas_caching_mode () const

bool get_agas_range_caching_mode () const

std::size_t get_agas_max_pending_refcnt_requests () const

bool load_application_configuration (char const *filename, error_code &ec =
                                         throws)

bool get_itt_notify_mode () const

bool enable_lock_detection () const

bool enable_global_lock_detection () const
```

```

bool enable_minimal_deadlock_detection() const
bool enable_spinlock_deadlock_detection() const
std::size_t get_spinlock_deadlock_detection_limit() const
std::size_t trace_depth() const
std::size_t get_os_thread_count() const
std::string get_cmd_line() const
std::ptrdiff_t get_default_stack_size() const
std::ptrdiff_t get_stack_size(threads::thread_stacksize stacksize) const
std::size_t get_thread_pool_size(char const *poolname) const
std::string get_endian_out() const
std::uint64_t get_max_inbound_message_size() const
std::uint64_t get_max_outbound_message_size() const
std::map<std::string, hpx::util::plugin::dll> &modules()

```

Public Members

runtime_mode **mode_**

Private Functions

```

std::ptrdiff_t init_stack_size(char const *entryname, char const *defaultvaluestr,
                      std::ptrdiff_t defaultvalue) const
std::ptrdiff_t init_small_stack_size() const
std::ptrdiff_t init_medium_stack_size() const
std::ptrdiff_t init_large_stack_size() const
std::ptrdiff_t init_huge_stack_size() const
void pre_initialize_ini()
void post_initialize_ini(std::string &hpx_ini_file, std::vector<std::string> const
&cmdline_ini_defs)
void pre_initialize_logging_ini()
void reconfigure()
void load_component_paths(std::vector<std::shared_ptr<plugins::plugin_registry_base>>
&plugin_registries, std::vector<std::shared_ptr<components::component_registry_
&component_registries, std::string const &com-
ponent_base_paths, std::string const &compo-
nent_path_suffixes, std::set<std::string> &compo-
nent_paths, std::map<std::string, filesystem::path> &base-
names)

```

```
void load_component_path(std::vector<std::shared_ptr<plugins::plugin_registry_base>>
    &plugin_registries, std::vector<std::shared_ptr<components::component_registry_base>>
    &component_registries, std::string const &path,
    std::set<std::string> &component_paths,
    std::map<std::string, filesystem::path> &basenames)
```

Private Members

```
std::string hpx_ini_file
std::vector<std::string> cmdline_ini_defs
std::vector<std::string> extra_static_ini_defs
std::uint32_t num_localities
std::uint32_t num_os_threads
std::ptrdiff_t small_stacksize
std::ptrdiff_t medium_stacksize
std::ptrdiff_t large_stacksize
std::ptrdiff_t huge_stacksize
bool need_to_call_pre_initialize
std::map<std::string, hpx::util::plugin::dll> modules_

namespace hpx
```

Enums

enum runtime_mode

A HPX runtime can be executed in two different modes: console mode and worker mode.

Values:

invalid = -1

console = 0

The runtime is the console locality.

worker = 1

The runtime is a worker locality.

connect = 2

The runtime is a worker locality connecting late

local = 3

The runtime is fully local.

default_ = 4

The runtime mode will be determined based on the command line arguments

last

Functions

```
char const *get_runtime_mode_name (runtime_mode state)
    Get the readable string representing the name of the given runtime_mode constant.

runtime_mode get_runtime_mode_from_name (std::string const &mode)
    Returns the internal representation (runtime_mode constant) from the readable string representing the name.

This represents the internal representation from the readable string representing the name.
```

Parameters

- mode: this represents the runtime mode

Defines

```
HPX_DECLARE_FACTORY_STATIC (name, base)
HPX_DEFINE_FACTORY_STATIC (module, name, base)
HPX_INIT_REGISTRY_MODULE_STATIC (name, base)
HPX_INIT_REGISTRY_FACTORY_STATIC (name, componentname, base)
HPX_INIT_REGISTRY_COMMANDLINE_STATIC (name, base)
HPX_INIT_REGISTRY_STARTUP_SHUTDOWN_STATIC (name, base)
namespace hpx
```

```
namespace components
```

Functions

```
bool &get_initial_static_loading ()
std::vector<static_factory_load_data_type> &get_static_module_data ()
void init_registry_module (static_factory_load_data_type const&)
bool get_static_factory (std::string const &instance, util::plugin::get_plugins_list_type &f)
void init_registry_factory (static_factory_load_data_type const&)
bool get_static_commandline (std::string const &instance,
                            util::plugin::get_plugins_list_type &f)
void init_registry_commandline (static_factory_load_data_type const&)
bool get_static_startup_shutdown (std::string const &instance,
                                  util::plugin::get_plugins_list_type &f)
void init_registry_startup_shutdown (static_factory_load_data_type const&)
struct static_factory_load_data_type
```

Public Members

```
char const *name  
hpx::util::plugin::get_plugins_list_type get_factory
```

runtime_local

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

HPX_REGISTER_STARTUP_SHUTDOWN_REGISTRY (*RegistryType, componentname*)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the components.

HPX_REGISTER_STARTUP_SHUTDOWN_REGISTRY_DYNAMIC (*RegistryType, componentname*)

HPX_REGISTER_STARTUP_SHUTDOWN_FUNCTIONS ()

This macro is used to define the required Hpx.Plugin entry point for the startup/shutdown registry. This macro has to be used in not more than one compilation unit of a component module.

HPX_REGISTER_STARTUP_SHUTDOWN_FUNCTIONS_DYNAMIC ()

```
namespace hpx
```

```
namespace components
```

```
struct component_startup_shutdown_base
```

#include <component_startup_shutdown_base.hpp> The component_startup_shutdown_base has to be used as a base class for all component startup/shutdown registries.

Public Functions

```
virtual ~component_startup_shutdown_base()
```

```
virtual bool get_startup_function (startup_function_type &startup, bool &pre_startup) = 0
```

Return any startup function for this component.

Return Returns *true* if the parameter *startup* has been successfully initialized with the startup function.

Parameters

- *startup*: [in, out] The module is expected to fill this function object with a reference to a startup function. This function will be executed by the runtime system during system startup.

```
virtual bool get_shutdown_function (shutdown_function_type &shutdown, bool &pre_shutdown) = 0
```

Return any shutdown function for this component.

Return Returns *true* if the parameter *shutdown* has been successfully initialized with the shutdown function.

Parameters

- `shutdown`: [in, out] The module is expected to fill this function object with a reference to a startup function. This function will be executed by the runtime system during system startup.

namespace hpx**Functions**`std::string get_config_entry (std::string const &key, std::string const &dflt)`

Retrieve the string value of a configuration entry given by `key`.

`std::string get_config_entry (std::string const &key, std::size_t dflt)`

Retrieve the integer value of a configuration entry given by `key`.

`void set_config_entry (std::string const &key, std::string const &value)`

Set the string value of a configuration entry given by `key`.

`void set_config_entry (std::string const &key, std::size_t value)`

Set the integer value of a configuration entry given by `key`.

`void set_config_entry_callback (std::string const &key, util::function_nonser<void> std::string const&, std::string const& > const &callback)`

Set the string value of a configuration entry given by `key`.

namespace hpx**Functions**`std::string diagnostic_information (exception_info const &xi)`

Extract the diagnostic information embedded in the given exception and return a string holding a formatted message.

The function `hpx::diagnostic_information` can be used to extract all diagnostic information stored in the given exception instance as a formatted string. This simplifies debug output as it composes the diagnostics into one, easy to use function call. This includes the name of the source file and line number, the sequence number of the OS-thread and the HPX-thread id, the locality id and the stack backtrace of the point where the original exception was thrown.

Return The formatted string holding all of the available diagnostic information stored in the given exception instance.

See `hpx::get_error_locality_id()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`, `hpx::get_error_function_name()`, `hpx::get_error_file_name()`, `hpx::get_error_line_number()`, `hpx::get_error_os_thread()`, `hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`, `hpx::get_error()`, `hpx::get_error_backtrace()`, `hpx::get_error_env()`, `hpx::get_error_what()`, `hpx::get_error_config()`, `hpx::get_error_state()`

Parameters

- `xi`: The parameter `e` will be inspected for all diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception_info`, `hpx::error_code`, `std::exception`, or `std::exception_ptr`.

Exceptions

- `std::bad_alloc`: (if any of the required allocation operations fail)

`std::uint32_t get_error_locality_id(hpx::exception_info const &xi)`

Return the locality id where the exception was thrown.

The function `hpx::get_error_locality_id` can be used to extract the diagnostic information element representing the locality id as stored in the given exception instance.

Return The locality id of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return `hpx::naming::invalid_locality_id`.

See `hpx::diagnostic_information()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`,
`hpx::get_error_function_name()`, `hpx::get_error_file_name()`, `hpx::get_error_line_number()`,
`hpx::get_error_os_thread()`, `hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`,
`hpx::get_error()`, `hpx::get_error_backtrace()`, `hpx::get_error_env()`, `hpx::get_error_what()`,
`hpx::get_error_config()`, `hpx::get_error_state()`

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception_info`, `hpx::error_code`, `std::exception`, or `std::exception_ptr`.

Exceptions

- `nothing`:

`std::string get_error_host_name(hpx::exception_info const &xi)`

Return the hostname of the locality where the exception was thrown.

The function `hpx::get_error_host_name` can be used to extract the diagnostic information element representing the host name as stored in the given exception instance.

Return The hostname of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return 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_backtrace (hpx::exception_info const &xi)`

Return the stack backtrace from the point the exception was thrown.

The function `hpx::get_error_backtrace` can be used to extract the diagnostic information element representing the stack backtrace collected at the point the exception was thrown.

Return The stack back trace from the point the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

See `hpx::diagnostic_information()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`,
`hpx::get_error_function_name()`, `hpx::get_error_file_name()`, `hpx::get_error_line_number()`,
`hpx::get_error_os_thread()`, `hpx::get_error_thread_id()`, `hpx::get_error_thread_description()`,
`hpx::get_error()`, `hpx::get_error_env()`, `hpx::get_error_what()`, `hpx::get_error_config()`,
`hpx::get_error_state()`

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: `hpx::exception_info`, `hpx::error_code`, `std::exception`, or `std::exception_ptr`.

Exceptions

- `std::bad_alloc`: (if one of the required allocations fails)

`std::size_t get_error_os_thread(hpx::exception_info const &xi)`

Return the sequence number of the OS-thread used to execute HPX-threads from which the exception was thrown.

The function `hpx::get_error_os_thread` can be used to extract the diagnostic information element representing the sequence number of the OS-thread as stored in the given exception instance.

Return The sequence number of the OS-thread used to execute the HPX-thread from which the exception was thrown. If the exception instance does not hold this information, the function will return `std::size_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)

namespace hpx

namespace util

Functions

`void may_attach_debugger (std::string const &category)`

Attaches a debugger if `category` is equal to the configuration entry `hpx.attach-debugger`.

namespace hpx

Functions

`std::uint32_t get_locality_id (error_code &ec = throws)`

Return the number of the locality this function is being called from.

This function returns the id of the current locality.

Note The returned value is zero based and its maximum value is smaller than the overall number of localities the current application is running on (as returned by `get_num_localities()`).

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

namespace hpx

Functions

`std::string get_locality_name()`

Return the name of the locality this function is called on.

This function returns the name for the locality on which this function is called.

Return This function returns the name for the locality on which the function is called. The name is retrieved from the underlying networking layer and may be different for different parcelports.

See `future<std::string> get_locality_name(naming::id_type const& id)`

`namespace hpx`

Functions

`std::uint32_t get_initial_num_localities()`

Return the number of localities which were registered at startup for the running application.

The function `get_initial_num_localities` returns the number of localities which were connected to the 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`

`hpx::future<std::uint32_t> get_num_localities()`

Asynchronously return the number of localities which are currently registered for the running application.

The function `get_num_localities` asynchronously returns the number of localities currently connected to the console. The returned future represents the actual result.

Note This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

See `hpx::find_all_localities, hpx::get_num_localities`

`std::uint32_t get_num_localities(launch::sync_policy, error_code &ec = throws)`

Return the number of localities which are currently registered for the running application.

The function `get_num_localities` returns the number of localities currently connected to the console.

Note This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

See `hpx::find_all_localities, hpx::get_num_localities`

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
namespace hpx
```

Functions

`std::size_t get_os_thread_count ()`

Return the number of OS-threads running in the runtime instance the current HPX-thread is associated with.

`std::size_t get_os_thread_count (threads::executor const &exec)`

Return the number of worker OS- threads used by the given executor to execute HPX threads.

This function returns the number of cores used to execute HPX threads for the given executor. If the function is called while no HPX runtime system is active, it will return zero. If the executor is not valid, this function will fall back to retrieving the number of OS threads used by HPX.

Parameters

- `exec`: [in] The executor to be used.

```
namespace hpx
```

Functions

`std::string get_thread_name ()`

Return the name of the calling thread.

This function returns the name of the calling thread. This name uniquely identifies the thread in the context of HPX. If the function is called while no HPX runtime system is active, the result will be “<unknown>”.

```
namespace hpx
```

```
namespace util
```

```
class interval_timer
```

Public Functions

`HPX_NON_COPYABLE (interval_timer)`

`interval_timer ()`

`interval_timer (util::function_nonser<bool>`

`> const &fstd::int64_t microsecs, std::string const &description = "", bool pre_shutdown = false`

`interval_timer (util::function_nonser<bool>`

`> const &util::function_nonser<void> const &on_termstd::int64_t microsecs, std::string const &description = "", bool pre_shutdown = false`

`interval_timer (util::function_nonser<bool>`

`> const &fhp::chrono::steady_duration const &rel_time, char const *description = "", bool pre_shutdown = false`

```

interval_timer(util::function_nonsr<bool>
  > const &util::function_nonsr<void> const &on_termhpx::chrono::steady_duration
  const &rel_time, char const *description = "", bool pre_shutdown = false

~interval_timer()

bool start(bool evaluate = true)

bool stop(bool terminate = false)

bool restart(bool evaluate = true)

bool is_started() const

bool is_terminated() const

std::int64_t get_interval() const

void change_interval(std::int64_t new_interval)

void change_interval(hpx::chrono::steady_duration const &new_interval)

```

Private Members

```

std::shared_ptr<detail::interval_timer> timer_

namespace hpx

```

```
namespace runtime_local
```

Enums

```
enum os_thread_type
```

Types of kernel threads registered with the runtime.

Values:

```
unknown = -1
```

```
main_thread = 0
```

kernel thread represents main thread

```
worker_thread
```

kernel thread is used to schedule HPX threads

```
io_thread
```

kernel thread can be used for IO operations

```
timer_thread
```

kernel is used by timer operations

```
parcel_thread
```

kernel is used by networking operations

```
custom_thread
```

kernel is registered by the application

Functions

```
std::string get_os_thread_type_name(os_thread_type type)
    Return a human-readable name representing one of the kernel thread types.

struct os_thread_data
    #include <os_thread_type.hpp> Registration data for kernel threads that is maintained by the runtime
    internally
```

Public Members

```
std::string label_
    name used for thread registration

std::thread::id id_
    thread id of corresponding kernel thread

std::uint64_t native_handle_
    the threads native handle

os_thread_type type_
    HPX thread type.
```

```
namespace hpx
```

```
namespace util
```

```
class pool_timer
```

Public Functions

```
HPX_NON_COPYABLE(pool_timer)

pool_timer()

pool_timer(util::function_nonser<bool>
    > const &util::function_nonser<void> const &on_termstd::string const &description = "", 
    bool pre_shutdown = true

~pool_timer()

bool start(hpx::chrono::steady_duration const &time_duration, bool evaluate = false)

bool stop()

bool is_started() const

bool is_terminated() const
```

Private Members

```
std::shared_ptr<detail::pool_timer> timer_
namespace hpx
```

Functions

void report_error (std::size_t num_thread, std::exception_ptr const &e)

The function report_error reports the given exception to the console.

void report_error (std::exception_ptr const &e)

The function report_error reports the given exception to the console.

```
namespace hpx
```

```
namespace threads
```

Functions

template<typename F, typename ...Ts>

util::invoke_result<F, Ts...>::type **run_as_hpx_thread (F const &f, Ts&&... vs)**

```
namespace hpx
```

```
namespace threads
```

Functions

template<typename F, typename ...Ts>

hpx::future<typename util::invoke_result<F, Ts...>::type> **run_as_os_thread (F &&f, Ts&&... vs)**

```
namespace hpx
```

Functions

void set_error_handlers ()

```
class runtime
```

Public Types

using notification_policy_type = threads::policies::callback_notifier

Generate a new notification policy instance for the given thread name prefix

using hpx_main_function_type = int ()

The *hpx_main_function_type* is the default function type usable as the main HPX thread function.

using hpx_errorssink_function_type = void (std::uint32_t, std::string const &)

Public Functions

```
virtual notification_policy_type get_notification_policy(char const *prefix, run-
time_local::os_thread_type
type)

state get_state() const

void set_state(state s)

runtime(hpx::util::runtime_configuration &rtcfg, bool initialize)
Construct a new HPX runtime instance.

virtual ~runtime()
The destructor makes sure all HPX runtime services are properly shut down before exiting.

void on_exit(util::function_nonser<void>
> const &fManage list of functions to call on exit.

void starting()
Manage runtime ‘stopped’ state.

void stopping()
Call all registered on_exit functions.

bool stopped() const
This accessor returns whether the runtime instance has been stopped.

hpx::util::runtime_configuration &get_config()
access configuration information

hpx::util::runtime_configuration const &get_config() const

std::size_t get_instance_number() const

util::thread_mapper &get_thread_mapper()
Return a reference to the internal PAPI thread manager.

threads::topology const &get_topology() const

virtual int run(util::function_nonser<hpx_main_function_type> const &func)
Run the HPX runtime system, use the given function for the main thread and block waiting for all
threads to finish.
```

Note The parameter *func* is optional. If no function is supplied, the runtime system will simply wait for the shutdown action without explicitly executing any main thread.

Return This function will return the value as returned as the result of the invocation of the function object given by the parameter *func*.

Parameters

- *func*: [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the typedef *hpx_main_function_type*. This parameter is optional and defaults to none main thread function, in which case all threads have to be scheduled explicitly.

virtual int run ()

Run the HPX runtime system, initially use the given number of (OS) threads in the thread-manager and block waiting for all threads to finish.

Return This function will always return 0 (zero).

virtual void rethrow_exception ()

Rethrow any stored exception (to be called after *stop()*)

virtual int start (util::function_nonser<hpx_main_function_type> const &func, bool blocking = false)

Start the runtime system.

Return If a blocking is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter *func*. Otherwise it will return zero.

Parameters

- *func*: [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the typedef *hpx_main_function_type*.
- *blocking*: [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function runtime::start will call runtime::wait internally.

virtual int start (bool blocking = false)

Start the runtime system.

Return If a blocking is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter *func*. Otherwise it will return zero.

Parameters

- *blocking*: [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function runtime::start will call runtime::wait internally .

virtual int wait ()

Wait for the shutdown action to be executed.

Return This function will return the value as returned as the result of the invocation of the function object given by the parameter *func*.

virtual void stop (bool blocking = true)

Initiate termination of the runtime system.

Parameters

- *blocking*: [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

virtual int suspend ()

Suspend the runtime system.

virtual int resume ()

Resume the runtime system.

```
virtual int finalize(double)

virtual bool is_networking_enabled()
    Return true if networking is enabled.

virtual hpx::threads::threadmanager &get_thread_manager()
    Allow access to the thread manager instance used by the HPX runtime.

virtual std::string here() const
    Returns a string of the locality endpoints (usable in debug output)

virtual bool report_error(std::size_t num_thread, std::exception_ptr const &e, bool terminate_all = true)
    Report a non-recoverable error to the runtime system.
```

Parameters

- num_thread: [in] The number of the operating system thread the error has been detected in.
- e: [in] This is an instance encapsulating an exception which lead to this function call.

```
virtual bool report_error(std::exception_ptr const &e, bool terminate_all = true)
    Report a non-recoverable error to the runtime system.
```

Note This function will retrieve the number of the current shepherd thread and forward to the report_error function above.

Parameters

- e: [in] This is an instance encapsulating an exception which lead to this function call.

```
virtual void add_pre_startup_function(startup_function_type f)
    Add a function to be executed inside a HPX thread before hpx_main but guaranteed to be executed before any startup function registered with add_startup_function.
```

Note The difference to a startup function is that all pre-startup functions will be (system-wide) executed before any startup function.

Parameters

- f: The function ‘f’ will be called from inside a HPX thread before hpx_main is executed. This is very useful to setup the runtime environment of the application (install performance counters, etc.)

```
virtual void add_startup_function(startup_function_type f)
    Add a function to be executed inside a HPX thread before hpx_main
```

Parameters

- f: The function ‘f’ will be called from inside a HPX thread before hpx_main is executed. This is very useful to setup the runtime environment of the application (install performance counters, etc.)

```
virtual void add_pre_shutdown_function(shutdown_function_type f)
    Add a function to be executed inside a HPX thread during hpx::finalize, but guaranteed before any of the shutdown functions is executed.
```

Note The difference to a shutdown function is that all pre-shutdown functions will be (system-wide) executed before any shutdown function.

Parameters

- `f`: The function ‘`f`’ will be called from inside a HPX thread while `hpx::finalize` is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

virtual void add_shutdown_function (*shutdown_function_type f*)

Add a function to be executed inside a HPX thread during `hpx::finalize`

Parameters

- `f`: The function ‘`f`’ will be called from inside a HPX thread while `hpx::finalize` is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

virtual hpx::util::io_service_pool *get_thread_pool (char const *name)

Access one of the internal thread pools (io_service instances) HPX is using to perform specific tasks. The three possible values for the argument `name` are “main_pool”, “io_pool”, “parcel_pool”, and “timer_pool”. For any other argument value the function will return zero.

**virtual bool register_thread (char const *name, std::size_t num = 0, bool service_thread
= true, error_code &ec = throws)**

Register an external OS-thread with HPX.

This function should be called from any OS-thread which is external to HPX (not created by HPX), but which needs to access HPX functionality, such as setting a value on a promise or similar.

‘main’, ‘io’, ‘timer’, ‘parcel’, ‘worker’

Note The function will compose a thread name of the form ‘<name>-thread#<num>’ which is used to register the thread. It is the user’s responsibility to ensure that each (composed) thread name is unique. HPX internally uses the following names for the threads it creates, do not reuse those:

Parameters

- `name`: [in] The name to use for thread registration.
- `num`: [in] The sequence number to use for thread registration. The default for this parameter is zero.
- `service_thread`: [in] The thread should be registered as a service thread. The default for this parameter is ‘true’. Any service threads will be pinned to cores not currently used by any of the HPX worker threads.

Note This function should be called for each thread exactly once. It will fail if it is called more than once.

Return This function will return whether the requested operation succeeded or not.

virtual bool unregister_thread ()

Unregister an external OS-thread with HPX.

This function will unregister any external OS-thread from HPX.

Note This function should be called for each thread exactly once. It will fail if it is called more than once. It will fail as well if the thread has not been registered before (see `register_thread`).

Return This function will return whether the requested operation succeeded or not.

**virtual runtime_local::os_thread_data get_os_thread_data (std::string const &label)
const**

Access data for a given OS thread that was previously registered by `register_thread`.

**virtual bool enumerate_os_threads (util::function_nonser<bool> runtime_local::os_thread_data
const &> const &f const**

Enumerate all OS threads that have registered with the runtime.

```
notification_policy_type::on_startstop_type on_start_func () const
notification_policy_type::on_startstop_type on_stop_func () const
notification_policy_type::on_error_type on_error_func () const
notification_policy_type::on_startstop_type on_start_func (notification_policy_type::on_startstop_type&&)
notification_policy_type::on_startstop_type on_stop_func (notification_policy_type::on_startstop_type&&)
notification_policy_type::on_error_type on_error_func (notification_policy_type::on_error_type&&)

virtual std::uint32_t get_locality_id(error_code &ec) const
virtual std::size_t get_num_worker_threads() const
virtual std::uint32_t get_num_localities(hpx::launch::sync_policy, error_code &ec) const
virtual std::uint32_t get_initial_num_localities() const
virtual hpx::future<std::uint32_t> get_num_localities() const
virtual std::string get_locality_name() const
virtual std::uint32_t assign_cores(std::string const&, std::uint32_t)
virtual std::uint32_t assign_cores()
```

Public Static Functions

```
static std::uint64_t get_system_uptime()
    Return the system uptime measure on the thread executing this call.
```

Protected Types

```
using on_exit_type = std::vector<util::function_nonser<void ()>>;
```

Protected Functions

```
runtime(hpx::util::runtime_configuration &rtcfg)
void set_notification_policies(notification_policy_type &&notifier,
                                threads::detail::network_background_callback_type
                                network_background_callback)
void init()
    Common initialization for different constructors.
void init_global_data()
void deinit_global_data()
threads::thread_result_type run_helper(util::function_nonser<runtime::hpx_main_function_type>
const &func, int &result, bool call_startup_functions)
void wait_helper(std::mutex &mtx, std::condition_variable &cond, bool &running)
```

Protected Attributes

```
on_exit_type on_exit_functions_
std::mutex mtx_
hpx::util::runtime_configuration rtdcfg_
long instance_number_
std::unique_ptr<util::thread_mapper> thread_support_
threads::topology &topology_
std::atomic<state> state_
notification_policy_type::on_startstop_type on_start_func_
notification_policy_type::on_startstop_type on_stop_func_
notification_policy_type::on_error_type on_error_func_
int result_
std::exception_ptr exception_
notification_policy_type main_pool_notifier_
util::io_service_pool main_pool_
notification_policy_type notifier_
std::unique_ptr<hpx::threads::threadmanager> thread_manager_
```

Protected Static Attributes

```
std::atomic<int> instance_number_counter_
```

Private Functions

void stop_helper (bool *blocking*, *std::condition_variable &cond*, *std::mutex &mtx*)
 Helper function to stop the runtime.

Parameters

- *blocking*: [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

```
void deinit_tss_helper (char const *context, std::size_t num)
void init_tss_ex (char const *context, runtime_local::os_thread_type type, std::size_t local_thread_num, std::size_t global_thread_num, char const *pool_name, char const *postfix, bool service_thread, error_code &ec)
void init_tss_helper (char const *context, runtime_local::os_thread_type type, std::size_t local_thread_num, std::size_t global_thread_num, char const *pool_name, char const *postfix, bool service_thread)
void notify_finalize ()
```

```
void wait_finalize()  
void call_startup_functions(bool pre_startup)
```

Private Members

```
std::list<startup_function_type> pre_startup_functions_  
std::list<startup_function_type> startup_functions_  
std::list<shutdown_function_type> pre_shutdown_functions_  
std::list<shutdown_function_type> shutdown_functions_  
bool stop_called_  
bool stop_done_  
std::condition_variable wait_condition_
```

namespace threads

Functions

```
char const *get_stack_size_name(std::ptrdiff_t size)  
Returns the stack size name.
```

Get the readable string representing the given stack size constant.

Parameters

- *size*: this represents the stack size

```
std::ptrdiff_t get_default_stack_size()  
Returns the default stack size.
```

Get the default stack size in bytes.

```
std::ptrdiff_t get_stack_size(thread_stacksize)  
Returns the stack size corresponding to the given stack size enumeration.  
Get the stack size corresponding to the given stack size enumeration.
```

Parameters

- *size*: this represents the stack size

namespace util

Functions

```
bool retrieve_commandline_arguments(hpx::program_options::options_description  
                                     const &app_options,  
                                     hpx::program_options::variables_map &vm)
```

```
bool retrieve_commandline_arguments(std::string const &appname,  
                                     hpx::program_options::variables_map &vm)
```

namespace hpx

Functions

`bool register_thread(runtime *rt, char const *name, error_code &ec = throws)`

Register the current kernel thread with HPX, this should be done once for each external OS-thread intended to invoke HPX functionality. Calling this function more than once will return false.

`void unregister_thread(runtime *rt)`

Unregister the thread from HPX, this should be done once in the end before the external thread exists.

`runtime_local::os_thread_data get_os_thread_data(std::string const &label)`

Access data for a given OS thread that was previously registered by `register_thread`. This function must be called from a thread that was previously registered with the runtime.

`bool enumerate_os_threads(util::function_nonser<bool> os_thread_data const & > const &f)` Enumerate all OS threads that have registered with the runtime.

`std::size_t get_runtime_instance_number()`

Return the runtime instance number associated with the runtime instance the current thread is running in.

`bool register_on_exit(util::function_nonser<void> > const &f)`

Register a function to be called during system shutdown.

`bool is_starting()`

Test whether the runtime system is currently being started.

This function returns whether the runtime system is currently being started or not, e.g. whether the current state of the runtime system is `hpx::state_startup`

Note This function needs to be executed on a HPX-thread. It will return false otherwise.

`bool tolerate_node_faults()`

Test if HPX runs in fault-tolerant mode.

This function returns whether the runtime system is running in fault-tolerant mode

`bool is_running()`

Test whether the runtime system is currently running.

This function returns whether the runtime system is currently running or not, e.g. whether the current state of the runtime system is `hpx::state_running`

Note This function needs to be executed on a HPX-thread. It will return false otherwise.

`bool is_stopped()`

Test whether the runtime system is currently stopped.

This function returns whether the runtime system is currently stopped or not, e.g. whether the current state of the runtime system is `hpx::state_stopped`

Note This function needs to be executed on a HPX-thread. It will return false otherwise.

`bool is_stopped_or_shutting_down()`

Test whether the runtime system is currently being shut down.

This function returns whether the runtime system is currently being shut down or not, e.g. whether the current state of the runtime system is `hpx::state_stopped` or `hpx::state_shutdown`

Note This function needs to be executed on a HPX-thread. It will return false otherwise.

`std::size_t get_num_worker_threads()`

Return the number of worker OS- threads used to execute HPX threads.

This function returns the number of OS-threads used to execute HPX threads. If the function is called while no HPX runtime system is active, it will return zero.

`std::uint64_t get_system_uptime()`

Return the system uptime measure on the thread executing this call.

This function returns the system uptime measured in nanoseconds for the thread executing this call. If the function is called while no HPX runtime system is active, it will return zero.

`namespace hpx`

`namespace parallel`

`namespace execution`

Enums

`enum service_executor_type`

Values:

`io_thread_pool`

Selects creating a service executor using the I/O pool of threads

`parcel_thread_pool`

Selects creating a service executor using the parcel pool of threads

`timer_thread_pool`

Selects creating a service executor using the timer pool of threads

`main_thread`

Selects creating a service executor using the main thread

`struct io_pool_executor : public service_executor`

Public Functions

`io_pool_executor()`

`struct main_pool_executor : public service_executor`

Public Functions

```
main_pool_executor()
struct parcel_pool_executor : public service_executor
```

Public Functions

```
parcel_pool_executor(char const *name_suffix = "-tcp")
struct service_executor : public service_executor
```

Public Functions

```
service_executor(service_executor_type t, char const *name_suffix = "")
struct timer_pool_executor : public service_executor
```

Public Functions

```
timer_pool_executor()
namespace hpx
```

Typedefs

typedef util::unique_function<void ()> shutdown_function_type

The type of a function which is registered to be executed as a shutdown or pre-shutdown function.

Functions

void register_pre_shutdown_function(shutdown_function_type f)

Add a function to be executed by a HPX thread during `hpx::finalize()` but guaranteed before any shutdown function is executed (system-wide)

Any of the functions registered with `register_pre_shutdown_function` are guaranteed to be executed by an HPX thread during the execution of `hpx::finalize()` before any of the registered shutdown functions are executed (see: `hpx::register_shutdown_function()`).

Note If this function is called while the pre-shutdown functions are being executed, or after that point, it will raise a `invalid_status` exception.

See `hpx::register_shutdown_function()`

Parameters

- `f`: [in] The function to be registered to run by an HPX thread as a pre-shutdown function.

```
void register_shutdown_function(shutdown_function_type f)
```

Add a function to be executed by a HPX thread during `hpx::finalize()` but guaranteed after any pre-shutdown function is executed (system-wide)

Any of the functions registered with `register_shutdown_function` are guaranteed to be executed by an HPX thread during the execution of `hpx::finalize()` after any of the registered pre-shutdown functions are executed (see: `hpx::register_pre_shutdown_function()`).

Note If this function is called while the shutdown functions are being executed, or after that point, it will raise a `invalid_status` exception.

See `hpx::register_pre_shutdown_function()`

Parameters

- `f`: [in] The function to be registered to run by an HPX thread as a shutdown function.

```
namespace hpx
```

Typedefs

```
typedef util::unique_function_nonser<void ()> startup_function_type
```

The type of a function which is registered to be executed as a startup or pre-startup function.

Functions

```
void register_pre_startup_function(startup_function_type f)
```

Add a function to be executed by a HPX thread before `hpx_main` but guaranteed before any startup function is executed (system-wide).

Any of the functions registered with `register_pre_startup_function` are guaranteed to be executed by an HPX thread before any of the registered startup functions are executed (see `hpx::register_startup_function()`).

This function is one of the few API functions which can be called before the runtime system has been fully initialized. It will automatically stage the provided startup function to the runtime system during its initialization (if necessary).

Note If this function is called while the pre-startup functions are being executed or after that point, it will raise a `invalid_status` exception.

Parameters

- `f`: [in] The function to be registered to run by an HPX thread as a pre-startup function.

See `hpx::register_startup_function()`

```
void register_startup_function(startup_function_type f)
```

Add a function to be executed by a HPX thread before `hpx_main` but guaranteed after any pre-startup function is executed (system-wide).

Any of the functions registered with `register_startup_function` are guaranteed to be executed by an HPX thread after any of the registered pre-startup functions are executed (see: `hpx::register_pre_startup_function()`), but before `hpx_main` is being called.

This function is one of the few API functions which can be called before the runtime system has been fully initialized. It will automatically stage the provided startup function to the runtime system during its initialization (if necessary).

Note If this function is called while the startup functions are being executed or after that point, it will raise a `invalid_status` exception.

Parameters

- `f`: [in] The function to be registered to run by an HPX thread as a startup function.

See `hpx::register_pre_startup_function()`

namespace hpx

namespace threads

Functions

`bool threadmanager_is(state st)`

`bool threadmanager_is_at_least(state st)`

namespace hpx

Functions

`threads::policies::callback_notifier::on_startstop_type get_thread_on_start_func()`

Retrieve the currently installed start handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered start function chains into the previous one (see `register_thread_on_start_func`).

Return The currently installed error handler function.

Note This function can be called before the HPX runtime is initialized.

`threads::policies::callback_notifier::on_startstop_type get_thread_on_stop_func()`

Retrieve the currently installed stop handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered stop function chains into the previous one (see `register_thread_on_stop_func`).

Return The currently installed error handler function.

Note This function can be called before the HPX runtime is initialized.

`threads::policies::callback_notifier::on_error_type get_thread_on_error_func()`

Retrieve the currently installed error handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function

reference, thus the caller needs to make sure any newly registered error function chains into the previous one (see *register_thread_on_error_func*).

Return The currently installed error handler function.

Note This function can be called before the HPX runtime is initialized.

```
threads::policies::callback_notifier::on_startstop_type register_thread_on_start_func(threads::policies::callback_notifier::on_startstop_type&&f)
```

Set the currently installed start handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered start function chains into the previous one (see *get_thread_on_start_func*).

Return The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

Note This function can be called before the HPX runtime is initialized.

Parameters

- f: The function to install as the new start handler.

```
threads::policies::callback_notifier::on_startstop_type register_thread_on_stop_func(threads::policies::callback_notifier::on_startstop_type&&f)
```

Set the currently installed stop handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered stop function chains into the previous one (see *get_thread_on_stop_func*).

Return The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

Note This function can be called before the HPX runtime is initialized.

Parameters

- f: The function to install as the new stop handler.

```
threads::policies::callback_notifier::on_error_type register_thread_on_error_func(threads::policies::callback_notifier::on_error_type&&f)
```

Set the currently installed error handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered error function chains into the previous one (see *get_thread_on_error_func*).

Return The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

Note This function can be called before the HPX runtime is initialized.

Parameters

- f: The function to install as the new error handler.

```
namespace hpx
```

```
namespace util

class thread_mapper
```

Public Types

```
using callback_type = detail::thread_mapper_callback_type
```

Public Functions

```
HPX_NON_COPYABLE(thread_mapper)

thread_mapper()
~thread_mapper()

std::uint32_t register_thread(char const *label, runtime_local::os_thread_type type)
bool unregister_thread()

std::uint32_t get_thread_index(std::string const &label) const
std::uint32_t get_thread_count() const

bool register_callback(std::uint32_t tix, callback_type const&)
bool revoke_callback(std::uint32_t tix)

std::thread::id get_thread_id(std::uint32_t tix) const
std::uint64_t get_thread_native_handle(std::uint32_t tix) const
std::string const &get_thread_label(std::uint32_t tix) const
runtime_local::os_thread_type get_thread_type(std::uint32_t tix) const

bool enumerate_os_threads(util::function_nonser<bool> os_thread_data const&
    > const &f const

os_thread_data get_os_thread_data(std::string const &label) const
```

Public Static Attributes

```
constexpr std::uint32_t invalid_index = std::uint32_t(-1)
constexpr std::uint64_t invalid_tid = std::uint64_t(-1)
```

Private Types

```
using mutex_type = hpx::lcos::local::spinlock
using thread_map_type = std::vector<detail::os_thread_data>
using label_map_type = std::map<std::string, std::size_t>
```

Private Members

```
mutex_type mtx_
thread_map_type thread_map_
label_map_type label_map_

namespace hpx

namespace resource
```

Functions

`std::size_t get_num_thread_pools()`

Return the number of thread pools currently managed by the *resource_partitioner*

`std::size_t get_num_threads()`

Return the number of threads in all thread pools currently managed by the *resource_partitioner*

`std::size_t get_num_threads(std::string const &pool_name)`

Return the number of threads in the given thread pool currently managed by the *resource_partitioner*

`std::size_t get_num_threads(std::size_t pool_index)`

Return the number of threads in the given thread pool currently managed by the *resource_partitioner*

`std::size_t get_pool_index(std::string const &pool_name)`

Return the internal index of the pool given its name.

`std::string const &get_pool_name(std::size_t pool_index)`

Return the name of the pool given its internal index.

`threads::thread_pool_base &get_thread_pool(std::string const &pool_name)`

Return the name of the pool given its name.

`threads::thread_pool_base &get_thread_pool(std::size_t pool_index)`

Return the thread pool given its internal index.

`bool pool_exists(std::string const &pool_name)`

Return true if the pool with the given name exists.

`bool pool_exists(std::size_t pool_index)`

Return true if the pool with the given index exists.

`namespace threads`

Functions

```
std::int64_t get_thread_count (thread_schedule_state state = thread_schedule_state::unknown)
```

The function `get_thread_count` returns the number of currently known threads.

Note If `state == unknown` this function will not only return the number of currently existing threads, but will add the number of registered task descriptions (which have not been converted into threads yet).

Parameters

- `state`: [in] This specifies the thread-state for which the number of threads should be retrieved.

```
std::int64_t get_thread_count (thread_priority priority, thread_schedule_state state = thread_schedule_state::unknown)
```

The function `get_thread_count` returns the number of currently known threads.

Note If `state == unknown` this function will not only return the number of currently existing threads, but will add the number of registered task descriptions (which have not been converted into threads yet).

Parameters

- `priority`: [in] This specifies the thread-priority for which the number of threads should be retrieved.
- `state`: [in] This specifies the thread-state for which the number of threads should be retrieved.

```
std::int64_t get_idle_core_count ()
```

The function `get_idle_core_count` returns the number of currently idling threads (cores).

```
mask_type get_idle_core_mask ()
```

The function `get_idle_core_mask` returns a bit-mask representing the currently idling threads (cores).

```
bool enumerate_threads (util::function_nonser<bool> thread_id_type
```

> const &f, thread_schedule_state state = thread_schedule_state::unknownThe function `enumerate_threads` will invoke the given function `f` for each thread with a matching thread state.

Parameters

- `f`: [in] The function which should be called for each matching thread. Returning ‘false’ from this function will stop the enumeration process.
- `state`: [in] This specifies the thread-state for which the threads should be enumerated.

```
namespace hpx
```

```
namespace util
```

```
namespace debug
```

Functions

```
std::vector<hpx::threads::thread_id_type> get_task_ids (hpx::threads::thread_schedule_state  
state  
=  
hpx::threads::thread_schedule_state::suspended)  
  
std::vector<hpx::threads::thread_data*> get_task_data (hpx::threads::thread_schedule_state  
state  
=  
hpx::threads::thread_schedule_state::suspended)  
  
std::string suspended_task_backtraces ()
```

schedulers

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

namespace threads

namespace policies

TypeDefs

```
using default_local_priority_queue_scheduler_terminated_queue = lockfree_fifo  
  
template<typename Mutex = std::mutex, typename PendingQueuing = lockfree_fifo, typename StagedQueuing  
class local_priority_queue_scheduler : public scheduler_base  
#include <local_priority_queue_scheduler.hpp> The local_priority_queue_scheduler maintains  
exactly one queue of work items (threads) per OS thread, where this OS thread pulls its next work  
from. Additionally it maintains separate queues: several for high priority threads and one for low  
priority threads. High priority threads are executed by the first N OS threads before any other  
work is executed. Low priority threads are executed by the last OS thread whenever no other  
work is available.
```

Public Types

```
typedef std::false_type has_periodic_maintenance  
typedef thread_queue<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing> thread_queue_type  
typedef init_parameter init_parameter_type
```

Public Functions

```

local_priority_queue_scheduler(init_parameter_type const &init, bool deferred_initialization = true)
~local_priority_queue_scheduler()

void abort_all_suspended_threads()

bool cleanup_terminated(bool delete_all)
bool cleanup_terminated(std::size_t num_thread, bool delete_all)
void create_thread(thread_init_data &data, thread_id_ref_type *id, error_code &ec)
bool get_next_thread(std::size_t num_thread, bool running,
                      threads::thread_id_ref_type &thrd, bool enable_stealing)
    Return the next thread to be executed, return false if none is available

void schedule_thread(threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint,
                      bool allow_fallback = false, thread_priority priority = thread_priority::normal)
    Schedule the passed thread.

void schedule_thread_last(threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint,
                           bool allow_fallback = false, thread_priority priority = thread_priority::normal)
void destroy_thread(threads::thread_data *thrd)
    Destroy the passed thread as it has been terminated.

std::int64_t get_queue_length(std::size_t num_thread = std::size_t(-1)) const
std::int64_t get_thread_count(thread_schedule_state state = thread_schedule_state::unknown, thread_priority priority = thread_priority::default_, std::size_t num_thread = std::size_t(-1), bool = false) const

bool is_core_idle(std::size_t num_thread) const
bool enumerate_threads(util::function_nonser<bool> thread_id_type > const &f, thread_schedule_state state = thread_schedule_state::unknown const)
bool wait_or_add_new(std::size_t num_thread, bool running, std::int64_t &idle_loop_count, bool enable_stealing, std::size_t &added)
    This is a function which gets called periodically by the thread manager to allow for maintenance tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to be terminated (i.e. no more work has to be done).

void on_start_thread(std::size_t num_thread)
void on_stop_thread(std::size_t num_thread)
void on_error(std::size_t num_thread, std::exception_ptr const &e)
void reset_thread_distribution()

```

Public Static Functions

```
static std::string get_scheduler_name()
```

Protected Attributes

```
std::atomic<std::size_t> curr_queue_
detail::affinity_data const &affinity_data_
const std::size_t num_queues_
const std::size_t num_high_priority_queues_
thread_queue_type low_priority_queue_
std::vector<util::cache_line_data<thread_queue_type*>> queues_
std::vector<util::cache_line_data<thread_queue_type*>> high_priority_queues_
std::vector<util::cache_line_data<std::vector<std::size_t>>> victim_threads_
struct init_parameter
```

Public Functions

```
template<>
init_parameter(std::size_t num_queues, detail::affinity_data const &affinity_data,
               std::size_t num_high_priority_queues = std::size_t(-1),
               thread_queue_init_parameters thread_queue_init = {}, char const *description = "local_priority_queue_scheduler")
template<>
init_parameter(std::size_t num_queues, detail::affinity_data const &affinity_data,
               char const *description)
```

Public Members

```
template<>
std::size_t num_queues_
template<>
std::size_t num_high_priority_queues_
template<>
thread_queue_init_parameters thread_queue_init_
template<>
detail::affinity_data const &affinity_data_
template<>
char const *description_
```

```
namespace hpx
```

```
namespace threads
```

```
namespace policies
```

TypeDefs

```
using default_local_queue_scheduler_terminated_queue = lockfree_fifo
template<typename Mutex = std::mutex, typename PendingQueuing = lockfree_fifo, typename StagedQueuing>
class local_queue_scheduler : public scheduler_base
#include <local_queue_scheduler.hpp> The local_queue_scheduler maintains exactly one queue
of work items (threads) per OS thread, where this OS thread pulls its next work from.
```

Public Types

```
typedef std::false_type has_periodic_maintenance
```

```
typedef thread_queue<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing> thread_queue_type
```

```
typedef init_parameter init_parameter_type
```

Public Functions

```
local_queue_scheduler(init_parameter_type const &init, bool deferred_initialization = true)
```

```
virtual ~local_queue_scheduler()
```

```
void abort_all_suspended_threads()
```

```
bool cleanup_terminated(bool delete_all)
```

```
bool cleanup_terminated(std::size_t num_thread, bool delete_all)
```

```
void create_thread(thread_init_data &data, thread_id_ref_type *id, error_code &ec)
```

```
virtual bool get_next_thread(std::size_t num_thread, bool running,
threads::thread_id_ref_type &thrd, bool)
```

Return the next thread to be executed, return false if none is available

```
void schedule_thread(threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint,
bool allow_fallback, thread_priority = thread_priority::normal)
```

Schedule the passed thread.

```
void schedule_thread_last(threads::thread_id_ref_type thrd,
threads::thread_schedule_hint schedulehint, bool allow_fallback, thread_priority = thread_priority::normal)
```

```
void destroy_thread(threads::thread_data *thrd)
```

Destroy the passed thread as it has been terminated.

```
std::int64_t get_queue_length(std::size_t num_thread = std::size_t(-1)) const
```

```
std::int64_t get_thread_count(thread_schedule_state state =
thread_schedule_state::unknown, thread_priority priority = thread_priority::default_, std::size_t num_thread =
std::size_t(-1), bool = false) const
```

```
bool is_core_idle(std::size_t num_thread) const
bool enumerate_threads(util::function_nonsig<bool> thread_id_type
    > const &f, thread_schedule_state state = thread_schedule_state::unknown const
virtual bool wait_or_add_new(std::size_t num_thread, bool running, std::int64_t
    &idle_loop_count, bool, std::size_t &added)
This is a function which gets called periodically by the thread manager to allow for maintenance
tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to
be terminated (i.e. no more work has to be done).
void on_start_thread(std::size_t num_thread)
void on_stop_thread(std::size_t num_thread)
void on_error(std::size_t num_thread, std::exception_ptr const &e)
```

Public Static Functions

```
static std::string get_scheduler_name()
```

Protected Attributes

```
std::vector<thread_queue_type*> queues_
std::atomic<std::size_t> curr_queue_
detail::affinity_data const &affinity_data_
mask_type steals_in_numa_domain_
mask_type steals_outside_numa_domain_
std::vector<mask_type> numa_domain_masks_
std::vector<mask_type> outside_numa_domain_masks_
struct init_parameter
```

Public Functions

```
template<>
init_parameter(std::size_t num_queues, detail::affinity_data const &affinity_data,
    thread_queue_init_parameters thread_queue_init = {}, char const
    *description = "local_queue_scheduler")

template<>
init_parameter(std::size_t num_queues, detail::affinity_data const &affinity_data,
    char const *description)
```

Public Members

```
template<>
std::size_t num_queues_

template<>
thread_queue_init_parameters thread_queue_init_

template<>
detail::affinity_data const &affinity_data_

template<>
char const *description_

namespace hpx
```

```
namespace threads
```

```
namespace policies
```

```
struct concurrentqueue_fifo
```

```
template<typename T>
struct apply
```

Public Types

```
template<>
using type = moodycamel_fifo_backend<T>
```

```
struct lockfree_fifo
```

```
template<typename T>
struct apply
```

Public Types

```
template<>
using type = lockfree_fifo_backend<T>
```

```
template<typename T>
struct lockfree_fifo_backend
```

Public Types

```
template<>
using container_type = boost::lockfree::queue<T, hpx::util::aligned_allocator<T>>

template<>
using value_type = T

template<>
using reference = T&

template<>
using const_reference = T const&

template<>
using rvalue_reference = T&&

template<>
using size_type = std::uint64_t
```

Public Functions

```
lockfree_fifo_backend (size_type initial_size = 0, size_type = size_type(-1))

bool push (const_reference val, bool = false)

bool push (rvalue_reference val, bool = false)

bool pop (reference val, bool = true)

bool empty ()
```

Private Members

```
container_type queue_

template<typename T>
struct moodycamel_fifo_backend
```

Public Types

```
template<>
using container_type = hpx::concurrency::ConcurrentQueue<T>

template<>
using value_type = T

template<>
using reference = T&

template<>
using const_reference = T const&

template<>
using rvalue_reference = T&&

template<>
using size_type = std::uint64_t
```

Public Functions

```
moodycamel_fifo_backend(size_type initial_size = 0, size_type = size_type(-1))
    bool push(const_reference val, bool = false)
    bool push(rvalue_reference val, bool = false)
    bool pop(reference val, bool = true)
    bool empty()
```

Private Members

```
container_type queue_
```

Defines

```
QUEUE HOLDER NUMA DEBUG
namespace hpx
```

Functions

```
static hpx::debug::enable_print<QUEUE HOLDER NUMA DEBUG> hpx::nq_deb("QH_NUMA")
namespace threads

namespace policies

template<typename QueueType>
struct queue_holder_numa
```

Public Types

```
template<>
using ThreadQueue = queue_holder_thread<QueueType>
template<>
using mutex_type = typename QueueType::mutex_type
```

Public Functions

```
queue_holder_numa()
~queue_holder_numa()
void init(std::size_t domain, std::size_t queues)
std::size_t size() const
ThreadQueue *thread_queue(std::size_t id) const
```

```
bool get_next_thread_HP (std::size_t qidx, threads::thread_id_ref_type &thrd, bool
                           stealing, bool core_stealing)

bool get_next_thread (std::size_t qidx, threads::thread_id_ref_type &thrd, bool stealing,
                      bool core_stealing)

bool add_new_HP (ThreadQueue *receiver, std::size_t qidx, std::size_t &added, bool steal-
                           ing, bool allow_stealing)

bool add_new (ThreadQueue *receiver, std::size_t qidx, std::size_t &added, bool stealing,
                           bool allow_stealing)

std::size_t get_new_tasks_queue_length() const

std::int64_t get_thread_count(thread_schedule_state state =
                           thread_schedule_state::unknown, thread_priority priority =
                           thread_priority::default_) const

void abort_all_suspended_threads()

bool enumerate_threads(util::function_nonser<bool> thread_id_type
                           > const &f, thread_schedule_state state const

void increment_num_pending_misses(std::size_t = 1)

void increment_num_pending_accesses(std::size_t = 1)

void increment_num_stolen_from_pending(std::size_t = 1)

void increment_num_stolen_from_staged(std::size_t = 1)

void increment_num_stolen_to_pending(std::size_t = 1)

void increment_num_stolen_to_staged(std::size_t = 1)

bool dump_suspended_threads(std::size_t, std::int64_t&, bool)

void debug_info()

void on_start_thread(std::size_t)

void on_stop_thread(std::size_t)

void on_error(std::size_t, std::exception_ptr const&)
```

Public Members

```
std::size_t num_queues_
std::size_t domain_
std::vector<ThreadQueue*> queues_
```

Defines

```
QUEUE HOLDER THREAD DEBUG
namespace hpx
```

Functions

```
static hpx::debug::enable_print<QUEUE HOLDER THREAD DEBUG> hpx::tq_deb ("QH_THRD")
namespace threads

namespace policies
```

Enums

```
enum [anonymous]
  Values:
    max_thread_count = 1000
enum [anonymous]
  Values:
    round_robin_rollover = 1
```

Functions

```
std::size_t fast_mod (std::size_t const input, std::size_t const ceil)
template<typename QueueType>
struct queue_holder_thread
```

Public Types

```
template<>
using thread_holder_type = queue_holder_thread<QueueType>
template<>
using mutex_type = std::mutex
typedef std::unique_lock<mutex_type> scoped_lock
template<>
using thread_heap_type = std::list<thread_id_type, util::internal_allocator<thread_id_type>>
template<>
using task_description = thread_init_data
template<>
using thread_map_type = std::unordered_set<thread_id_type, std::hash<thread_id_type>, std::equal_to<th
```

Public Functions

```
queue_holder_thread(QueueType *bp_queue, QueueType *hp_queue, QueueType *np_queue, QueueType *lp_queue, std::size_t domain, std::size_t queue, std::size_t thread_num, std::size_t owner, const thread_queue_init_parameters &init)

~queue_holder_thread()

bool owns_bp_queue() const

bool owns_hp_queue() const

bool owns_np_queue() const

bool owns_lp_queue() const

std::size_t worker_next(std::size_t const workers) const

void schedule_thread(threads::thread_id_ref_type thrd, thread_priority priority, bool other_end = false)

bool cleanup_terminated(std::size_t thread_num, bool delete_all)

void create_thread(thread_init_data &data, thread_id_ref_type *tid, std::size_t thread_num, error_code &ec)

void create_thread_object(threads::thread_id_ref_type &tid, threads::thread_init_data &data)

void recycle_thread(thread_id_type tid)

void add_to_thread_map(threads::thread_id_type tid)

void remove_from_thread_map(threads::thread_id_type tid, bool dealloc)

bool get_next_thread_HP(threads::thread_id_ref_type &thrd, bool stealing, bool check_new)

bool get_next_thread(threads::thread_id_ref_type &thrd, bool stealing)

std::size_t add_new_HP(std::int64_t add_count, thread_holder_type *addfrom, bool stealing)

std::size_t add_new(std::int64_t add_count, thread_holder_type *addfrom, bool stealing)

std::size_t get_queue_length()

std::size_t get_thread_count_staged(thread_priority priority) const

std::size_t get_thread_count_pending(thread_priority priority) const

std::size_t get_thread_count(thread_schedule_state state = thread_schedule_state::unknown, thread_priority priority = thread_priority::default_) const

void destroy_thread(threads::thread_data *thrd, std::size_t thread_num, bool xthread)
    Destroy the passed thread as it has been terminated.

void abort_all_suspended_threads()
```

```

bool enumerate_threads (util::function_nonsr<bool> thread_id_type
    > const &f, thread_schedule_state state = thread_schedule_state::unknown const
    void debug_info ()
    void debug_queues (const char *prefix)

```

Public Members

```

QueueType *const bp_queue_
QueueType *const hp_queue_
QueueType *const np_queue_
QueueType *const lp_queue_
const std::size_t domain_index_
const std::size_t queue_index_
const std::size_t thread_num_
const std::size_t owner_mask_
util::cache_line_data<mutex_type> thread_map_mtx_
thread_heap_type thread_heap_small_
thread_heap_type thread_heap_medium_
thread_heap_type thread_heap_large_
thread_heap_type thread_heap_huge_
thread_heap_type thread_heap_nostack_
util::cache_line_data<std::tuple<std::size_t, std::size_t>> rollover_counters_
thread_map_type thread_map_
util::cache_line_data<std::atomic<std::int32_t>> thread_map_count_
terminated_items_type terminated_items_
util::cache_line_data<std::atomic<std::int32_t>> terminated_items_count_
thread_queue_init_parameters parameters_

```

Public Static Functions

```

static void deallocate (threads::thread_data *p)

```

Public Static Attributes

```
util::internal_allocator<threads::thread_data> thread_alloc_
struct queue_data_print
```

Public Functions

```
template<>
queue_data_print(const queue_holder_thread *q)
```

Public Members

```
template<>
const queue_holder_thread *q
```

Friends

```
std::ostream &operator<<(std::ostream &os, const queue_data_print &d)
struct queue_mc_print
```

Public Functions

```
template<>
queue_mc_print(const QueueType *const q)
```

Public Members

```
template<>
const QueueType *const q
```

Friends

```
std::ostream &operator<<(std::ostream &os, const queue_mc_print &d)
```

Defines

```
SHARED_PRIORITY_SCHEDULER_DEBUG
SHARED_PRIORITY_QUEUE_SCHEDULER_API
namespace hpx
```

Typedefs

```
using print_onoff = hpx::debug::enable_print<SHARED_PRIORITY_SCHEDULER_DEBUG>
using print_on = hpx::debug::enable_print<false>
```

Functions

```
static print_onoff hpx::spq_deb("SPQUEUE")
static print_on hpx::spq_arr("SPQUEUE")
```

namespace threads

namespace policies

Typedefs

```
using default_shared_priority_queue_scheduler_terminated_queue = lockfree_fifo
struct core_ratios
```

Public Functions

```
core_ratios(std::size_t high_priority, std::size_t normal_priority, std::size_t
low_priority)
```

Public Members

```
std::size_t high_priority
std::size_t normal_priority
std::size_t low_priority
```

```
template<typename Mutex = std::mutex, typename PendingQueuing = concurrentqueue_fifo, typename Terminator = std::atomic<int>>
class shared_priority_queue_scheduler : public scheduler_base
#include <shared_priority_queue_scheduler.hpp> The shared_priority_queue_scheduler maintains a set of high, normal, and low priority queues. For each priority level there is a core/queue ratio which determines how many cores share a single queue. If the high priority core/queue ratio is 4 the first 4 cores will share a single high priority queue, the next 4 will share another one and so on. In addition, the shared_priority_queue_scheduler is NUMA-aware and takes NUMA scheduling hints into account when creating and scheduling work.
```

Warning: PendingQueuing lifo causes lockup on termination

Public Types

```
template<>
using has_periodic_maintenance = std::false_type

template<>
using thread_queue_type = thread_queue_mc<Mutex, PendingQueuing, PendingQueuing, TerminatedQu
```

```
template<>
using thread_holder_type = queue_holder_thread<thread_queue_type>

typedef init_parameter init_parameter_type
```

Public Functions

```
shared_priority_queue_scheduler (init_parameter const &init)

virtual ~shared_priority_queue_scheduler ()

void set_scheduler_mode (scheduler_mode mode)

void abort_all_suspended_threads ()

std::size_t local_thread_number ()

bool cleanup_terminated (bool delete_all)

bool cleanup_terminated (std::size_t, bool delete_all)

void create_thread (thread_init_data &data, thread_id_ref_type *thrd, error_code &ec)

template<typename T>
bool steal_by_function (std::size_t domain, std::size_t q_index, bool steal numa, bool
                      steal core, thread_holder_type *origin, T &var, const char
                      *prefix, util::function_nonser<bool> std::size_t, std::size_t,
                      thread_holder_type*, T&, bool, bool
                      > operation_HP, util::function_nonser<bool> std::size_t, std::size_t, thread_holder_type*, T&,
                      bool, bool> operation
```

```
virtual bool get_next_thread (std::size_t thread_num, bool running,
                           threads::thread_id_ref_type &thrd, bool enable_stealing)
```

Return the next thread to be executed, return false if none available.

```
virtual bool wait_or_add_new (std::size_t, bool, std::int64_t&, bool, std::size_t
                           &added)
```

Return the next thread to be executed, return false if none available.

```
void schedule_thread (threads::thread_id_ref_type thrd, threads::thread_schedule_hint
                      schedulehint, bool allow_fallback, thread_priority priority =
                      thread_priority::normal)
```

Schedule the passed thread.

```
void schedule_thread_last (threads::thread_id_ref_type thrd,
                           threads::thread_schedule_hint schedulehint,
                           bool allow_fallback, thread_priority priority =
                           thread_priority::normal)
```

Put task on the back of the queue : not yet implemented just put it on the normal queue for now

```

void destroy_thread(threads::thread_data *thrd)

std::int64_t get_queue_length(std::size_t thread_num = std::size_t(-1)) const

std::int64_t get_thread_count(thread_schedule_state state = thread_schedule_state::unknown, thread_priority priority = thread_priority::default_, std::size_t thread_num = std::size_t(-1), bool = false) const

bool is_core_idle(std::size_t num_thread) const

bool enumerate_threads(util::function_nonser<bool> thread_id_type > const &f, thread_schedule_state state = thread_schedule_state::unknown) const

void on_start_thread(std::size_t local_thread)

void on_stop_thread(std::size_t thread_num)

void on_error(std::size_t thread_num, std::exception_ptr const&)

```

Public Static Functions

```
static std::string get_scheduler_name()
```

Protected Types

```
typedef queue_holder numa<thread_queue_type> numa_queues
```

Protected Attributes

```

std::array<std::size_t, HPX_HAVE_MAX_NUMA_DOMAIN_COUNT> q_counts_
std::array<std::size_t, HPX_HAVE_MAX_NUMA_DOMAIN_COUNT> q_offset_
std::array<numa_queues, HPX_HAVE_MAX_NUMA_DOMAIN_COUNT> numa_holder_
std::vector<std::size_t> d_lookup_
std::vector<std::size_t> q_lookup_
core_ratios cores_per_queue_
bool round_robin_
bool steal_hp_first_
bool numa_stealing_
bool core_stealing_
std::size_t num_workers_
std::size_t num_domains_
detail::affinity_data const &affinity_data_
const thread_queue_init_parameters queue_parameters_
std::mutex init_mutex
bool initialized_

```

```
    bool debug_init_
    std::atomic<std::size_t> thread_init_counter_
    std::size_t pool_index_
struct init_parameter
```

Public Functions

```
template<>
init_parameter(std::size_t num_worker_threads, const core_ratios
&cores_per_queue, detail::affinity_data const &affinity_data,
const thread_queue_init_parameters &thread_queue_init, char
const *description = "shared_priority_queue_scheduler")

template<>
init_parameter(std::size_t num_worker_threads, const core_ratios
&cores_per_queue, detail::affinity_data const &affinity_data,
char const *description)
```

Public Members

```
template<>
std::size_t num_worker_threads_

template<>
core_ratios cores_per_queue_

template<>
thread_queue_init_parameters thread_queue_init_

template<>
detail::affinity_data const &affinity_data_

template<>
char const *description_

namespace hpx
```

```
namespace threads
```

```
namespace policies
```

Typedefs

```
using default_static_priority_queue_scheduler_terminated_queue = lockfree_fifo

template<typename Mutex = std::mutex, typename PendingQueueing = lockfree_fifo, typename StagedQueueing
class static_priority_queue_scheduler: public hpx::threads::policies::local_priority_queue_scheduler
#include <static_priority_queue_scheduler.hpp> The static_priority_queue_scheduler maintains
exactly one queue of work items (threads) per OS thread, where this OS thread pulls its next work
from. Additionally it maintains separate queues: several for high priority threads and one for low
priority threads. High priority threads are executed by the first N OS threads before any other
```

work is executed. Low priority threads are executed by the last OS thread whenever no other work is available. This scheduler does not do any work stealing.

Public Types

```
template<>
using base_type = local_priority_queue_scheduler<Mutex, PendingQueuing, StagedQueuing, TerminatedQu
```

```
template<>
using init_parameter_type = typename base_type::init_parameter_type
```

Public Functions

```
static_priority_queue_scheduler(init_parameter_type const &init, bool de-
ferred_initialization = true)
```

```
void set_scheduler_mode(scheduler_mode mode)
```

Public Static Functions

```
static std::string get_scheduler_name()
```

```
namespace hpx
```

```
namespace threads
```

```
namespace policies
```

Typedefs

```
using default_static_queue_scheduler_terminated_queue = lockfree_fifo
```

```
template<typename Mutex = std::mutex, typename PendingQueuing = lockfree_fifo, typename StagedQueuing
```

```
class static_queue_scheduler : public hpx::threads::policies::local_queue_scheduler<std::mutex, lockfr
```

```
#include <static_queue_scheduler.hpp> The local_queue_scheduler maintains exactly one queue
```

```
of work items (threads) per OS thread, where this OS thread pulls its next work from.
```

Public Types

```
typedef local_queue_scheduler<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing> base_type
```

Public Functions

```
static queue_scheduler(typename base_type::init_parameter_type const &init,
                      bool deferred_initialization = true)

void set_scheduler_mode(scheduler_mode mode)

bool get_next_thread(std::size_t num_thread, bool, threads::thread_id_ref_type &thrd,
                     bool)
    Return the next thread to be executed, return false if none is available

bool wait_or_add_new(std::size_t num_thread, bool running, std::int64_t
                     &idle_loop_count, bool, std::size_t &added)
    This is a function which gets called periodically by the thread manager to allow for maintenance
    tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to
    be terminated (i.e. no more work has to be done).
```

Public Static Functions

```
static std::string get_scheduler_name()

namespace hpx

namespace threads

namespace policies

template<typename Mutex, typename PendingQueuing, typename StagedQueuing, typename Terminated>
class thread_queue
```

Public Functions

```
bool cleanup_terminated_locked(bool delete_all = false)
    This function makes sure all threads which are marked for deletion (state is terminated) are
    properly destroyed.

    This returns ‘true’ if there are no more terminated threads waiting to be deleted.

bool cleanup_terminated(bool delete_all = false)

thread_queue(std::size_t queue_num = std::size_t(-1), thread_queue_init_parameters pa-
             rameters = {})

~thread_queue()

std::int64_t get_queue_length(std::memory_order          order           = =
                             std::memory_order_acquire) const

std::int64_t get_pending_queue_length(std::memory_order      order           = =
                                         std::memory_order_acquire) const

std::int64_t get_staged_queue_length(std::memory_order      order           = =
                                         std::memory_order_acquire) const

constexpr void increment_num_pending_misses(std::size_t = 1)
```

```

constexpr void increment_num_pending_accesses(std::size_t = 1)

constexpr void increment_num_stolen_from_pending(std::size_t = 1)

constexpr void increment_num_stolen_from_staged(std::size_t = 1)

constexpr void increment_num_stolen_to_pending(std::size_t = 1)

constexpr void increment_num_stolen_to_staged(std::size_t = 1)

void create_thread(thread_init_data &data, thread_id_ref_type *id, error_code &ec)

void move_work_items_from(thread_queue *src, std::int64_t count)

void move_task_items_from(thread_queue *src, std::int64_t count)

bool get_next_thread(threads::thread_id_ref_type &thrd, bool allow_stealing = false,
                     bool steal = false)
    Return the next thread to be executed, return false if none is available

void schedule_thread(threads::thread_id_ref_type thrd, bool other_end = false)
    Schedule the passed thread.

void destroy_thread(threads::thread_data *thrd)
    Destroy the passed thread as it has been terminated.

std::int64_t get_thread_count(thread_schedule_state state = thread_schedule_state::unknown) const
    Return the number of existing threads with the given state.

void abort_all_suspended_threads()

bool enumerate_threads(util::function_nonser<bool> thread_id_type
                      > const &f, thread_schedule_state state = thread_schedule_state::unknown const

bool wait_or_add_new(bool, std::size_t &added, bool steal = false)
    This is a function which gets called periodically by the thread manager to allow for maintenance
    tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to
    be terminated (i.e. no more work has to be done).

bool wait_or_add_new(bool running, std::size_t &added, thread_queue *addfrom, bool
                     steal = false)

bool dump_suspended_threads(std::size_t num_thread, std::int64_t &idle_loop_count,
                            bool running)

void on_start_thread(std::size_t)

void on_stop_thread(std::size_t)

void on_error(std::size_t, std::exception_ptr const&)

```

Public Static Functions

```
static void deallocate(threads::thread_data *p)
```

Protected Functions

```
template<typename Lock>
void create_thread_object(threads::thread_id_ref_type &thrd,
                         threads::thread_init_data &data, Lock &lk)
    std::size_t add_new(std::int64_t add_count, thread_queue *addfrom,
                         std::unique_lock<mutex_type> &lk, bool steal = false)
    bool add_new_always(std::size_t &added, thread_queue *addfrom,
                         std::unique_lock<mutex_type> &lk, bool steal = false)
void recycle_thread(thread_id_type thrd)
```

Protected Static Attributes

```
util::internal_allocator<typename thread_queue<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing>>
```

Private Types

```
template<>
using mutex_type = Mutex
template<>
using thread_map_type = std::unordered_set<thread_id_type, std::hash<thread_id_type>, std::equal_to<thread_id_type>>
template<>
using thread_heap_type = std::vector<thread_id_type, util::internal_allocator<thread_id_type>>
template<>
using thread_description_ptr = typename thread_id_ref_type::thread_repr*
template<>
using work_items_type = typename PendingQueuing::template apply<thread_description_ptr>::type
template<>
using task_items_type = typename StagedQueuing::template apply<task_description*>::type
template<>
using terminated_items_type = typename TerminatedQueuing::template apply<thread_data*>::type
```

Private Members

```
thread_queue_init_parameters parameters_
mutex_type mtx_
thread_map_type thread_map_
std::atomic<std::int64_t> thread_map_count_
work_items_type work_items_
terminated_items_type terminated_items_
```

```

std::atomic<std::int64_t> terminated_items_count_
task_items_type new_tasks_
thread_heap_type thread_heap_small_
thread_heap_type thread_heap_medium_
thread_heap_type thread_heap_large_
thread_heap_type thread_heap_huge_
thread_heap_type thread_heap_nostack_
util::cache_line_data<std::atomic<std::int64_t>> new_tasks_count_
util::cache_line_data<std::atomic<std::int64_t>> work_items_count_

struct task_description

```

Public Members

```

template<>
thread_init_data data

```

Defines

```
THREAD_QUEUE_MC_DEBUG
```

```
namespace hpx
```

Functions

```

static hpx::debug::enable_print<THREAD_QUEUE_MC_DEBUG> hpx::tqmc_deb("_TQ_MC_")
namespace threads

```

```
namespace policies
```

```

template<typename Mutex, typename PendingQueuing, typename StagedQueuing, typename TerminatedQueueing>
class thread_queue_mc

```

Public Types

```
typedef Mutex mutex_type
```

```
template<>
```

```
using thread_queue_type = thread_queue_mc<Mutex, PendingQueuing, StagedQueuing, TerminatedQueueing>;
```

```
template<>
```

```
using thread_heap_type = std::list<thread_id_type, util::internal_allocator<thread_id_type>>;
```

```
template<>
```

```
using task_description = thread_init_data
```

```
template<>
```

```
using thread_description = thread_data
```

```
typedef PendingQueueing::template apply<thread_id_ref_type>::type work_items_type
typedef concurrentqueue_fifo::apply<task_description>::type task_items_type
```

Public Functions

```
std::size_t add_new (std::int64_t add_count, thread_queue_type *addfrom, bool stealing)

thread_queue_mc (const thread_queue_init_parameters &parameters, std::size_t
queue_num = std::size_t(-1))

void set_holder (queue_holder_thread<thread_queue_type> *holder)

~thread_queue_mc ()

std::int64_t get_queue_length () const

std::int64_t get_queue_length_pending () const

std::int64_t get_queue_length_staged (std::memory_order order = std::memory_order_relaxed) const

std::int64_t get_thread_count () const

void create_thread (thread_init_data &data, thread_id_ref_type *id, error_code &ec)

bool get_next_thread (threads::thread_id_ref_type &thrd, bool other_end, bool
check_new = false)
    Return the next thread to be executed, return false if none is available

void schedule_work (threads::thread_id_ref_type thrd, bool other_end)
    Schedule the passed thread (put it on the ready work queue)

void on_start_thread (std::size_t)

void on_stop_thread (std::size_t)

void on_error (std::size_t, std::exception_ptr const&)
```

Public Members

```
thread_queue_init_parameters parameters_
const int queue_index_
queue_holder_thread<thread_queue_type> *holder_
task_items_type new_task_items_
work_items_type work_items_
util::cache_line_data<std::atomic<std::int32_t>> new_tasks_count_
util::cache_line_data<std::atomic<std::int32_t>> work_items_count_
```

serialization

See *Public API* for a list of names and headers that are part of the public HPX API.

```
template<typename T>
struct serialize_non_intrusive<T, std::enable_if_t<has_serialize_adl<T>::value>>
```

Public Static Functions

```
template<typename Archive>
static void call(Archive &ar, T &t, unsigned)
```

```
template<typename T>
struct serialize_brace_initialized<T, std::enable_if_t<has_struct_serialization<T>::value>>
```

Public Static Functions

```
template<typename Archive>
static void call(Archive &ar, T &t, unsigned)
```

```
namespace hpx
```

```
namespace serialization
```

```
class access
```

Public Static Functions

```
template<typename Archive, typename T>
static void serialize(Archive &ar, T &t, unsigned)
```

```
template<typename Archive, typename T>
static void save_base_object(Archive &ar, T const &t, unsigned)
```

```
template<typename Archive, typename T>
static void load_base_object(Archive &ar, T &t, unsigned)
```

```
template<typename T>
static std::string get_name(T const *t)
```

```
template<typename T>
class has_serialize
```

Public Static Attributes

```
constexpr bool value = decltype(test<T>(0))::value
```

Private Static Functions

```
template<typename T1>
static std::false_type test (...)
```

```
template<typename T1, typename = decltype(std::declval<typename std::remove_const<T1>::type&>().serialize(<T1>))>
static std::true_type test (int)
```

```
template<typename T>
class serialize_dispatcher
```

Public Types

```
template<>
using type = std::conditional_t<hpx::traits::is_intrusive_polymorphic_v<T>, intrusive_polymorphic, std::conditional_t<T::is_trivial, std::empty, T>>;
```

Public Static Functions

```
template<class Archive>
static void call (Archive&, T&, unsigned)
```

```
struct intrusive_polymorphic
```

Public Static Functions

```
template<>
static void call (hpx::serialization::input_archive &ar, T &t, unsigned)
```

```
template<>
static void call (hpx::serialization::output_archive &ar, T const &t, unsigned)
```

```
struct intrusive_usual
```

Public Static Functions

```
template<class Archive>
static void call (Archive &ar, T &t, unsigned)
```

```
struct non_intrusive
```

Public Static Functions

```
template<class Archive>
static void call (Archive &ar, T &t, unsigned)  
  
template<typename T>
class has_serialize_adl
```

Public Static Attributes

```
constexpr bool value = decltype(test<T>(0))::value
```

Private Static Functions

```
template<typename T1>
static std::false_type test (...)  
  
template<typename T1, typename = decltype(serialize(std::declval<hpx::serialization::output_archive&x::serialization::output_archive&static std::true_type test (int)  
  
template<typename T>
class has_struct_serialization
```

Public Static Functions

```
template<typename T1>
static std::false_type test (...)  
  
template<typename T1, typename = decltype(serialize_struct(std::declval<hpx::serialization::output_archive&x::serialization::output_archive&static std::true_type test (int)
```

Public Static Attributes

```
constexpr bool value = decltype(test<T>(0))::value  
  
template<typename T>
struct serialize_brace_initialized<T, std::enable_if_t<has_struct_serialization<T>::value>>
```

Public Static Functions

```
template<typename Archive>
static void call (Archive &ar, T &t, unsigned)  
  
template<typename T>
struct serialize_non_intrusive<T, std::enable_if_t<has_serialize_adl<T>::value>>
```

Public Static Functions

```
template<typename Archive>
static void call (Archive &ar, T &t, unsigned)
```

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<class T>
array<T> make_array (T *begin, std::size_t size)

template<typename Archive, typename T, std::size_t N>
void serialize (Archive &ar, std::array<T, N> &a, const unsigned int)

template<typename T>
output_archive &operator<< (output_archive &ar, array<T> t)

template<typename T>
input_archive &operator>> (input_archive &ar, array<T> t)

template<typename T>
output_archive &operator& (output_archive &ar, array<T> t)

template<typename T>
input_archive &operator& (input_archive &ar, array<T> t)

template<typename T, std::size_t N>
output_archive &operator<< (output_archive &ar, T (&t)[N])

template<typename T, std::size_t N>
input_archive &operator>> (input_archive &ar, T (&t)[N])

template<typename T, std::size_t N>
output_archive &operator& (output_archive &ar, T (&t)[N])

template<typename T, std::size_t N>
input_archive &operator& (input_archive &ar, T (&t)[N])

template<class T>
class array
```

Public Types

```
template<>
using value_type = T
```

Public Functions

```
array (value_type *t, std::size_t s)
value_type *address () const
std::size_t count () const
template<class Archive>
void serialize_optimized (Archive &ar, unsigned int, std::false_type)
void serialize_optimized (output_archive &ar, unsigned int, std::true_type)
void serialize_optimized (input_archive &ar, unsigned int, std::true_type)
template<class Archive>
void serialize (Archive &ar, unsigned int v)
```

Private Members

```
value_type *m_t
std::size_t m_element_count
template<typename Derived, typename Base>
struct base_object_type<Derived, Base, std::true_type>
```

Public Functions

```
base_object_type (Derived &d)
template<class Archive>
void save (Archive &ar, unsigned) const
template<class Archive>
void load (Archive &ar, unsigned)
HPX_SERIALIZATION_SPLIT_MEMBER()
```

Public Members

```
Derived &d_
namespace hpx

namespace serialization
```

Functions

```
template<typename Base, typename Derived>
base_object_type<Derived, Base> base_object (Derived &d)  
  
template<typename D, typename B>
output_archive &operator<< (output_archive &ar, base_object_type<D, B> t)  
  
template<typename D, typename B>
input_archive &operator>> (input_archive &ar, base_object_type<D, B> t)  
  
template<typename D, typename B>
output_archive &operator& (output_archive &ar, base_object_type<D, B> t)  
  
template<typename D, typename B>
input_archive &operator& (input_archive &ar, base_object_type<D, B> t)  
  
template<typename Derived, typename Base, typename Enable = typename hpx::traits::is_intrusive_polymorphic<Derived>
struct base_object_type
```

Public Functions

```
base_object_type (Derived &d)  
  
template<typename Archive>
void serialize (Archive &ar, unsigned)
```

Public Members

```
Derived &d_
```

```
template<typename Derived, typename Base>
struct base_object_type<Derived, Base, std::true_type>
```

Public Functions

```
base_object_type (Derived &d)  
  
template<class Archive>
void save (Archive &ar, unsigned) const  
  
template<class Archive>
void load (Archive &ar, unsigned)  
  
HPX_SERIALIZATION_SPLIT_MEMBER()
```

Public Members

Derived & d_

```
namespace hpx
```

```
namespace serialization
```

Enums

```
enum archive_flags
```

Values:

```
no_archive_flags = 0x00000000
```

```
enable_compression = 0x00002000
```

```
endian_big = 0x00004000
```

```
endian_little = 0x00008000
```

```
disable_array_optimization = 0x00010000
```

```
disable_data_chunking = 0x00020000
```

```
all_archive_flags = 0x0003e000
```

Functions

```
void reverse_bytes (std::size_t size, char *address)
```

```
template<typename Archive>
```

```
void save_binary (Archive &ar, void const *address, std::size_t count)
```

```
template<typename Archive>
```

```
void load_binary (Archive &ar, void *address, std::size_t count)
```

```
template<typename Archive>
```

```
std::size_t current_pos (const Archive &ar)
```

```
template<typename Archive>
```

```
struct basic_archive
```

Public Functions

```
virtual ~basic_archive ()
```

```
template<typename T>
```

```
void invoke (T &t)
```

```
bool enable_compression () const
```

```
bool endian_big () const
```

```
bool endian_little () const
```

```
bool disable_array_optimization () const
```

```
bool disable_data_chunking() const
std::uint32_t flags() const
bool is_preprocessing() const
std::size_t current_pos() const
void save_binary(void const *address, std::size_t count)
void load_binary(void *address, std::size_t count)
void reset()

template<typename T>
T &get_extra_data()

template<typename T>
T *try_get_extra_data()
```

Public Static Attributes

```
const std::uint64_t npos = std::uint64_t(-1)
```

Protected Functions

```
basic_archive(std::uint32_t flags)
basic_archive(basic_archive const&)
basic_archive &operator=(basic_archive const&)
```

Protected Attributes

```
std::uint32_t flags_
std::size_t size_
detail::extra_archive_data extra_data_

namespace hpx

namespace serialization

struct binary_filter
```

Public Functions

```

virtual void set_max_length(std::size_t size) = 0

virtual void save(void const *src, std::size_t src_count) = 0

virtual bool flush(void *dst, std::size_t dst_count, std::size_t &written) = 0

virtual std::size_t init_data(char const *buffer, std::size_t size, std::size_t buffer_size)
    = 0

virtual void load(void *dst, std::size_t dst_count) = 0

template<class T>
void serialize(T&, unsigned)

HPX_SERIALIZATION_POLYMORPHIC_ABSTRACT(binary_filter)
virtual ~binary_filter()

namespace hpx

```

```
namespace serialization
```

Functions

```

template<std::size_t N>
void serialize(input_archive &ar, std::bitset<N> &d, unsigned)

template<std::size_t N>
void serialize(output_archive &ar, std::bitset<N> const &bs, unsigned)

namespace hpx

```

```
namespace serialization
```

Functions

```

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
    hpx::traits::detail::size<0>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
    hpx::traits::detail::size<1>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
    hpx::traits::detail::size<2>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
    hpx::traits::detail::size<3>)

template<typename Archive, typename T>

```

```
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<4>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<5>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<6>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<7>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<8>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<9>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<10>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<11>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<12>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<13>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<14>)

template<typename Archive, typename T>
void serialize_struct(Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<15>)

template<typename Archive, typename T>
void serialize_struct(Archive &ar, T &t, const unsigned int version)

namespace hpx

namespace serialization
```

Functions

```
template<typename T>
void serialize(input_archive &ar, std::complex<T> &c, unsigned)

template<typename T>
void serialize(output_archive &ar, std::complex<T> const &c, unsigned)

namespace hpx

namespace serialization
```

struct erased_input_container

Subclassed by *hpx::serialization::input_container<Container>*

Public Functions

```
virtual ~erased_input_container()

virtual bool is_preprocessing() const

virtual void set_filter(binary_filter *filter) = 0

virtual void load_binary(void *address, std::size_t count) = 0

virtual void load_binary_chunk(void *address, std::size_t count) = 0
```

struct erased_output_container

Subclassed by *hpx::serialization::output_container<Container, Chunker>*

Public Functions

```
virtual ~erased_output_container()

virtual bool is_preprocessing() const

virtual void set_filter(binary_filter *filter) = 0

virtual void save_binary(void const *address, std::size_t count) = 0

virtual std::size_t save_binary_chunk(void const *address, std::size_t count) = 0

virtual void reset() = 0

virtual std::size_t get_num_chunks() const = 0

virtual void flush() = 0
```

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename T, typename Allocator>
void serialize(input_archive &ar, std::deque<T, Allocator> &d, unsigned)

template<typename T, typename Allocator>
void serialize(output_archive &ar, std::deque<T, Allocator> const &d, unsigned)

namespace hpx
```

```
    namespace serialization
```

Functions

```
template<typename Archive>
void save(Archive &ar, std::exception_ptr const &e, unsigned int)

template<typename Archive>
void load(Archive &ar, std::exception_ptr &e, unsigned int)
```

```
    namespace util
```

Enums

```
enum exception_type
  Values:
    unknown_exception = 0
    std_runtime_error = 1
    std_invalid_argument = 2
    std_out_of_range = 3
    std_logic_error = 4
    std_bad_alloc = 5
    std_bad_cast = 6
    std_bad_typeid = 7
    std_bad_exception = 8
    std_exception = 9
    boost_system_error = 10
    hpx_exception = 11
    hpx_thread_interrupted_exception = 12
    std_system_error = 14
```

```
    namespace hpx
```

```
        namespace serialization
```

```
struct input_archive : public hpx::serialization::basic_archive<input_archive>
```

Public Types

```
using base_type = basic_archive<input_archive>
```

Public Functions

```
template<typename Container>
input_archive(Container &buffer, std::size_t inbound_data_size = 0, const
               std::vector<serialization_chunk> *chunks = nullptr)

template<typename T>
void invoke_impl(T &t)

template<typename T>
std::enable_if_t<!std::is_integral_v<T> && !std::is_enum_v<T>> load(T &t)

template<typename T>
std::enable_if_t<std::is_integral_v<T> || std::is_enum_v<T>> load(T &t)

void load(float &f)

void load(double &d)

void load(char &c)

void load(bool &b)

std::size_t bytes_read() const

std::size_t current_pos() const
```

Private Functions

```
template<typename T>
void load_bitwise(T &t, std::false_type)

template<typename T>
void load_bitwise(T &t, std::true_type)

template<class T>
void load_nonintrusively_polymorphic(T &t, std::false_type)

template<class T>
void load_nonintrusively_polymorphic(T &t, std::true_type)

template<typename T>
void load_integral(T &val, std::false_type)

template<typename T>
void load_integral(T &val, std::true_type)

template<class Promoted>
void load_integral_impl(Promoted &l)
```

```
void load_binary(void *address, std::size_t count)
void load_binary_chunk(void *address, std::size_t count)
```

Private Members

```
std::unique_ptr<erased_input_container> buffer_
```

Friends

```
friend hpx::serialization::basic_archive< input_archive >
friend hpx::serialization::array
```

```
namespace hpx
```

```
namespace serialization
```

```
template<typename Container>
struct input_container : public hpx::serialization::erased_input_container
```

Public Functions

```
input_container(Container const &cont, std::size_t inbound_data_size)
input_container(Container const &cont, std::vector<serialization_chunk> const
    *chunks, std::size_t inbound_data_size)
void set_filter(binary_filter *filter)
void load_binary(void *address, std::size_t count)
void load_binary_chunk(void *address, std::size_t count)
```

Public Members

```
Container const &cont_
std::size_t current_
std::unique_ptr<binary_filter> filter_
std::size_t decompressed_size_
std::vector<serialization_chunk> const *chunks_
std::size_t current_chunk_
std::size_t current_chunk_size_
```

Private Types

```
template<>
using access_traits = traits::serialization_access_data<Container>
```

Private Functions

```
std::size_t get_chunk_size (std::size_t chunk) const
std::uint8_t get_chunk_type (std::size_t chunk) const
chunk_data get_chunk_data (std::size_t chunk) const
std::size_t get_num_chunks () const

namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename T, typename Allocator>
void serialize (input_archive &ar, std::list<T, Allocator> &ls, unsigned)

template<typename T, typename Allocator>
void serialize (output_archive &ar, const std::list<T, Allocator> &ls, unsigned)
```

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Key, typename Value>
void serialize (input_archive &ar, std::pair<Key, Value> &t, unsigned)

template<typename Key, typename Value>
void serialize (output_archive &ar, const std::pair<Key, Value> &t, unsigned)

template<typename Key, typename Value, typename Comp, typename Alloc>
void serialize (input_archive &ar, std::map<Key, Value, Comp, Alloc> &t, unsigned)

template<typename Key, typename Value, typename Comp, typename Alloc>
void serialize (output_archive &ar, std::map<Key, Value, Comp, Alloc> const &t, unsigned)
```

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename T>
void save (output_archive &ar, hpx::util::optional<T> const &o, unsigned)

template<typename T>
void load (input_archive &ar, hpx::util::optional<T> &o, unsigned)

namespace hpx

namespace serialization

struct output_archive : public hpx::serialization::basic_archive<output_archive>
```

Public Types

```
using base_type = basic_archive<output_archive>
```

Public Functions

```
template<typename Container>
output_archive (Container &buffer, std::uint32_t flags = 0U,
                 std::vector<serialization_chunk> *chunks = nullptr, binary_filter *filter =
                 nullptr)

std::size_t bytes_written () const

std::size_t get_num_chunks () const

std::size_t current_pos () const

void reset ()

void flush ()

bool is_preprocessing () const
```

Protected Functions

```
template<typename T>
void invoke_impl (T const &t)

template<typename T>
std::enable_if_t<!std::is_integral_v<T> && !std::is_enum_v<T>> save (T const &t)

template<typename T>
std::enable_if_t<std::is_integral_v<T> || std::is_enum_v<T>> save (T t)

void save (float f)

void save (double d)

void save (char c)
```

```

void save (bool b)
template<typename T>
void save_bitwise (T const &t, std::false_type)
template<typename T>
void save_bitwise (T const &t, std::true_type)
template<typename T>
void save_nonintrusively_polymorphic (T const &t, std::false_type)
template<typename T>
void save_nonintrusively_polymorphic (T const &t, std::true_type)
template<typename T>
void save_integral (T val, std::false_type)
template<typename T>
void save_integral (T val, std::true_type)
template<class Promoted>
void save_integral_impl (Promoted l)
void save_binary (void const *address, std::size_t count)
void save_binary_chunk (void const *address, std::size_t count)

```

Protected Attributes

```
std::unique_ptr<erased_output_container> buffer_
```

Private Static Functions

```
static std::uint32_t make_flags (std::uint32_t flags, std::vector<serialization_chunk> *chunks)
```

Friends

```

friend hpx::serialization::basic_archive< output_archive >
friend hpx::serialization::array
namespace hpx

namespace serialization

template<typename Container, typename Chunker>
struct filtered_output_container : public hpx::serialization::output_container<Container, Chunker>

```

Public Types

```
template<>
using access_traits = traits::serialization_access_data<Container>

template<>
using base_type = output_container<Container, Chunker>
```

Public Functions

```
filtered_output_container(Container &cont, std::vector<serialization_chunk>
                           *chunks = nullptr)

~filtered_output_container()

void flush()

void set_filter(binary_filter *filter)

void save_binary(void const *address, std::size_t count)

std::size_t save_binary_chunk(void const *address, std::size_t count)
```

Protected Attributes

```
std::size_t start_compressing_at_

binary_filter *filter_

template<typename Container, typename Chunker>
struct output_container : public hpx::serialization::erased_output_container
Subclassed by hpx::serialization::filtered_output_container< Container, Chunker >
```

Public Types

```
template<>
using access_traits = traits::serialization_access_data<Container>
```

Public Functions

```
output_container(Container &cont, std::vector<serialization_chunk> *chunks = nullptr)

~output_container()

void flush()

std::size_t get_num_chunks() const

void reset()

void set_filter(binary_filter*)

void save_binary(void const *address, std::size_t count)

std::size_t save_binary_chunk(void const *address, std::size_t count)

bool is_preprocessing() const
```

Protected Attributes

```
Container &cont_
std::size_t current_
Chunker chunker_
```

```
template<typename IArch, typename OArch, typename Char>
class basic_any<IArch, OArch, Char, std::true_type>
```

Public Functions

```
constexpr basic_any()

basic_any(basic_any const &x)

basic_any(basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>>::type>
basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::value>>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::value>>::type>
basic_any(std::in_place_type_t<T>, Ts&&... ts)

template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::value>>::type>
basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

~basic_any()

basic_any &operator=(basic_any const &x)

basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>>::type>
basic_any &operator=(T &&rhs)

basic_any &swap(basic_any &x)

std::type_info const &type() const

template<typename T>
T const &cast() const

bool has_value() const

void reset()

bool equal_to(basic_any const &rhs) const
```

Private Functions

```
basic_any &assign (basic_any const &x)
void load (IArch &ar, const unsigned version)
void save (OArch &ar, const unsigned version) const
HPX_SERIALIZATION_SPLIT_MEMBER ()
```

Private Members

```
detail::any::fxn_ptr_table<IArch, OArch, Char, std::true_type> *table
void *object
```

Private Static Functions

```
template<typename T, typename ...Ts>
static void new_object (void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static void new_object (void *&object, std::false_type, Ts&&... ts)
```

Friends

```
friend hpx::serialization::access
namespace hpx
```

Typedefs

```
using any = util::basic_any<serialization::input_archive, serialization::output_archive, char, std::true_type>
```

Functions

```
template<typename T, typename Char>
util::basic_any<serialization::input_archive, serialization::output_archive, Char> make_any (T &&t)
namespace util
```

Typedefs

```
using instead = basic_any<serialization::input_archive, serialization::output_archive, char, std::true_type>
using wany = basic_any<serialization::input_archive, serialization::output_archive, wchar_t, std::true_type>
```

Functions

```

template<typename T, typename Char, typename ...Ts>
basic_any<serialization::input_archive, serialization::output_archive, Char> make_any (Ts&&...
ts)

template<typename T, typename Char, typename U, typename ...Ts>
basic_any<serialization::input_archive, serialization::output_archive, Char> make_any (std::initializer_list<U>
il,
Ts&&...
ts)

template<typename T, typename Char> hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::ma

template<typename IArch, typename OArch, typename Char>
class basic_any<IArch, OArch, Char, std::true_type>

```

Public Functions

```

constexpr basic_any()

basic_any(basic_any const &x)

basic_any(basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value>::type* = nullptr>
basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::value>::type* = nullptr>

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename basic_any(Ts&&... ts)

template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<basic_any(U, Ts&&... ts)

~basic_any()

basic_any &operator=(basic_any const &x)

basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value>::type* = nullptr>
basic_any &operator=(T &&rhs)

basic_any &swap(basic_any &x)

std::type_info const &type() const

template<typename T>
T const &cast() const

bool has_value() const

void reset()

bool equal_to(basic_any const &rhs) const

```

Private Functions

```
basic_any &assign (basic_any const &x)
void load (IArch &ar, const unsigned version)
void save (OArch &ar, const unsigned version) const
HPX_SERIALIZATION_SPLIT_MEMBER()
```

Private Members

```
detail::any::fxn_ptr_table<IArch, OArch, Char, std::true_type> *table
void *object
```

Private Static Functions

```
template<typename T, typename ...Ts>
static void new_object (void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static void new_object (void *&object, std::false_type, Ts&&... ts)
```

Friends

```
friend hpx::util::hpx::serialization::access
struct hash_any
```

Public Functions

```
template<typename Char>
std::size_t operator() (const basic_any<serialization::input_archive, serialization::output_archive, Char, std::true_type> &elem) const
namespace hpx

namespace serialization
```

Enums

```
enum chunk_type
    Values:
        chunk_type_index = 0
        chunk_type_pointer = 1
```

Functions

```
serialization_chunk create_index_chunk (std::size_t index, std::size_t size)
serialization_chunk create_pointer_chunk (void const *pos, std::size_t size, std::uint64_t rkey = 0)
union chunk_data
```

Public Members

```
std::size_t index_
void const *cpos_
void *pos_
struct serialization_chunk
```

Public Members

```
chunk_data data_
std::size_t size_
std::uint64_t rkey_
std::uint8_t type_
```

Defines

```
HPX_SERIALIZATION_SPLIT_MEMBER()
HPX_SERIALIZATION_SPLIT_FREE (T)
HPX_SERIALIZATION_SPLIT_FREE_TEMPLATE (TEMPLATE, ARGS)
namespace hpx

namespace serialization
```

Functions

```
template<typename T>
output_archive &operator<< (output_archive &ar, T const &t)

template<typename T>
input_archive &operator>> (input_archive &ar, T &t)

template<typename T>
output_archive &operator& (output_archive &ar, T const &t)

template<typename T>
input_archive &operator& (input_archive &ar, T &t)
```

```
namespace hpx

namespace serialization

template<typename T, typename Allocator = std::allocator<T>>
class serialize_buffer
```

Public Types

```
enum init_mode
```

Values:

```
copy = 0
```

```
reference = 1
```

```
take = 2
```

```
template<>
```

```
using value_type = T
```

Public Functions

```
serialize_buffer(allocator_type const &alloc = allocator_type())
```

```
serialize_buffer(std::size_t size, allocator_type const &alloc = allocator_type())
```

```
serialize_buffer(T *data, std::size_t size, init_mode mode = copy, allocator_type const
&alloc = allocator_type())
```

```
template<typename Deallocator>
```

```
serialize_buffer(T *data, std::size_t size, allocator_type const &alloc, Deallocator
const &dealloc)
```

```
template<typename Deleter>
```

```
serialize_buffer(T *data, std::size_t size, init_mode mode, Deleter const &deleter,
allocator_type const &alloc = allocator_type())
```

```
template<typename Deleter>
```

```
serialize_buffer(T const *data, std::size_t size, init_mode mode, Deleter const
&deleter, allocator_type const &alloc = allocator_type())
```

```
template<typename Deallocator, typename Deleter>
```

```
serialize_buffer(T *data, std::size_t size, allocator_type const &alloc, Deallocator
const &, Deleter const &deleter)
```

```
serialize_buffer(T const *data, std::size_t size, allocator_type const &alloc = allo-
cator_type())
```

```
template<typename Deleter>
```

```
serialize_buffer(T const *data, std::size_t size, Deleter const &deleter, alloca-
tor_type const &alloc = allocator_type())
```

```
serialize_buffer(T const *data, std::size_t size, init_mode mode, allocator_type
const &alloc = allocator_type())
```

```
T *data()
```

```
T const *data() const
T *begin()
T *end()
T &operator[](std::size_t idx)
T operator[](std::size_t idx) const
buffer_type data_array() const
std::size_t size() const
void resize_norealloc(std::size_t newsize)
```

Private Types

```
template<>
using allocator_type = Allocator
template<>
using buffer_type = boost::shared_array<T>
```

Private Functions

```
template<typename Archive>
void save(Archive &ar, unsigned int const) const
template<typename Archive>
void load(Archive &ar, unsigned int const)
```

Private Members

```
buffer_type data_
std::size_t size_
Allocator alloc_
```

Private Static Functions

```
static void no_deleter(T*)
template<typename Deallocator>
static void deleter(T *p, Deallocator dealloc, std::size_t size)
```

Friends

```
friend hpx::serialization::hpx::serialization::access  
bool operator==(serialize_buffer const &rhs, serialize_buffer const &lhs)  
namespace hpx  
  
namespace serialization
```

Functions

```
template<typename T, typename Compare, typename Allocator>  
void serialize(input_archive &ar, std::set<T, Compare, Allocator> &set, unsigned)  
  
template<typename T, typename Compare, typename Allocator>  
void serialize(output_archive &ar, std::set<T, Compare, Allocator> const &set, unsigned)  
namespace hpx  
  
namespace serialization
```

Functions

```
template<typename T>  
void load(input_archive &ar, std::shared_ptr<T> &ptr, unsigned)  
  
template<typename T>  
void save(output_archive &ar, std::shared_ptr<T> const &ptr, unsigned)  
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Archive, typename ...Ts>  
void serialize(Archive &ar, std::tuple<Ts...> &t, unsigned int version)  
  
template<typename Archive>  
void serialize(Archive&, std::tuple<>&, unsigned int)
```

```
namespace hpx  
  
namespace serialization
```

Functions

```
template<typename Char, typename CharTraits, typename Allocator>
void serialize(input_archive &ar, std::basic_string<Char, CharTraits, Allocator> &s, unsigned)

template<typename Char, typename CharTraits, typename Allocator>
void serialize(output_archive &ar, std::basic_string<Char, CharTraits, Allocator> const &s, unsigned)

namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Archive, typename ...Ts>
void serialize(Archive &ar, hpx::tuple<Ts...> &t, unsigned int version)

template<typename Archive>
void serialize(Archive&, hpx::tuple<>&, unsigned)

template<typename Archive, typename ...Ts>
void load_construct_data(Archive &ar, hpx::tuple<Ts...> *t, unsigned int version)

template<typename Archive, typename ...Ts>
void save_construct_data(Archive &ar, hpx::tuple<Ts...> const *t, unsigned int version)
```

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename T>
void load(input_archive &ar, std::unique_ptr<T> &ptr, unsigned)

template<typename T>
void save(output_archive &ar, const std::unique_ptr<T> &ptr, unsigned)

namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Key, typename Value, typename Hash, typename KeyEqual, typename Alloc>
void serialize(input_archive &ar, std::unordered_map<Key, Value, Hash, KeyEqual, Alloc> &t,
               unsigned)

template<typename Key, typename Value, typename Hash, typename KeyEqual, typename Alloc>
void serialize(output_archive &ar, const std::unordered_map<Key, Value, Hash, KeyEqual,
               Alloc> &t, unsigned)

namespace hpx
```

```
    namespace serialization
```

Functions

```
template<typename T>
void serialize(input_archive &ar, std::valarray<T> &arr, int)

template<typename T>
void serialize(output_archive &ar, std::valarray<T> const &arr, int)

namespace hpx
```

```
    namespace serialization
```

Functions

```
template<typename ...Ts>
void save(output_archive &ar, std::variant<Ts...> const &v, unsigned)

template<typename ...Ts>
void load(input_archive &ar, std::variant<Ts...> &v, unsigned)

namespace hpx
```

```
    namespace serialization
```

Functions

```
template<typename Allocator>
void serialize(input_archive &ar, std::vector<bool, Allocator> &v, unsigned)

template<typename T, typename Allocator>
void serialize(input_archive &ar, std::vector<T, Allocator> &v, unsigned)

template<typename Allocator>
void serialize(output_archive &ar, std::vector<bool, Allocator> const &v, unsigned)

template<typename T, typename Allocator>
void serialize(output_archive &ar, std::vector<T, Allocator> const &v, unsigned)
```

Defines

```
MAKE_ARITY_FUNC (count)
namespace hpx
```

```
namespace traits
```

Functions

```
template<typename T, std::size_t N>
constexpr auto is_brace_constructible()

    hpx::traits::static_cast< T * >(nullptr)

template<typename T, std::size_t N>
constexpr auto is_paren_constructible()
```

Defines

```
HPX_IS_BITWISE_SERIALIZABLE (T)
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_bitwise_serializable_v = is_bitwise_serializable<T>::value
```

Defines

```
HPX_IS_NOT_BITWISE_SERIALIZABLE (T)
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_not_bitwise_serializable_v = true
```

Defines

```
HPX_TRAITS_NONINTRUSIVE_POLYMORPHIC (Class)
HPX_TRAITS_NONINTRUSIVE_POLYMORPHIC_TEMPLATE (TEMPLATE, ARG_LIST)
HPX_TRAITS_SERIALIZED_WITH_ID (Class)
HPX_TRAITS_SERIALIZED_WITH_ID_TEMPLATE (TEMPLATE, ARG_LIST)
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_intrusive_polymporphic_v = is_intrusive_polymporphic<T>::value

template<typename T>
constexpr bool is_nonintrusive_polymporphic_v = is_nonintrusive_polymporphic<T>::value

template<typename T>
constexpr bool is_serialized_with_id_v = is_serialized_with_id<T>::value

namespace hpx
```

```
namespace traits
```

```
template<typename Container>
struct default_serialization_access_data
    Subclassed by hpx::traits::serialization_access_data<Container>
```

Public Types

```
template<>
using preprocessing_only = std::false_type
```

Public Static Functions

```
static constexpr bool is_preprocessing()

static constexpr void write(Container&, std::size_t, std::size_t, void const*)

static bool flush(serialization::binary_filter*, Container&, std::size_t, std::size_t size,
                 std::size_t &written)

static constexpr void read(Container const&, std::size_t, std::size_t, void*)

static constexpr std::size_t init_data(Container const&, serialization::binary_filter*, std::size_t, std::size_t
                                         decompressed_size)

static constexpr void reset(Container&)
```

```
template<typename Container>
struct serialization_access_data : public hpx::traits::default_serialization_access_data<Container>
    Subclassed by hpx::traits::serialization_access_data<Container const >
```

Public Static Functions

```
static std::size_t size(Container const &cont)
static void resize(Container &cont, std::size_t count)
static void write(Container &cont, std::size_t count, std::size_t current, void const *address)
static bool flush(serialization::binary_filter *filter, Container &cont, std::size_t current,
                    std::size_t size, std::size_t &written)
static void read(Container const &cont, std::size_t count, std::size_t current, void *address)
static std::size_t init_data(Container const &cont, serialization::binary_filter *filter,
                               std::size_t current, std::size_t decompressed_size)
```

static_reinit

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_CORE_EXPORT_REINITIALIZABLE_STATIC
namespace hpx
```

```
namespace util
```

Variables

```
template<typename T, typename Tag = T, std::size_t N = 1>
struct HPX_CORE_EXPORT_REINITIALIZABLE_STATIC reinitializable_static

template<typename T, typename Tag, std::size_t N>
struct reinitializable_static
```

Public Types

```
typedef T value_type
typedef T &reference
typedef T const &const_reference
```

Public Functions

```
HPX_NON_COPYABLE(reinitializable_static)

reinitializable_static()

template<typename U>
reinitializable_static(U const &val)

operator reference()

operator const_reference() const

reference get(std::size_t item = 0)

const_reference get(std::size_t item = 0) const
```

Private Types

```
typedef std::add_pointer<value_type>::type pointer
typedef std::aligned_storage<sizeof(value_type), std::alignment_of<value_type>::value>::type storage_type
```

Private Static Functions

```
static void default_construct()

template<typename U>
static void value_construct(U const &v)

static void destruct()

static void default_constructor()

template<typename U>
static void value_constructor(U const *pv)

static pointer get_address(std::size_t item)
```

Private Static Attributes

```
reinitializable_static<T, Tag, N>::storage_type data_
std::once_flag constructed_

namespace hpx

namespace util
```

Functions

```
void reinit_register(util::function_nonser<void>
    > const &constructutil::function_nonser<void> const &destruct)

void reinit_construct()

void reinit_destruct()
```

string_util

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace string_util
```

Functions

```
template<typename CharT, class Traits, class Alloc>
void to_lower(std::basic_string<CharT, Traits, Alloc> &s)
```

```
namespace hpx
```

```
namespace string_util
```

Functions

```
template<typename CharT, typename Traits, typename Allocator>
detail::is_any_of_pred<CharT, Traits, Allocator> is_any_of(std::basic_string<CharT, Traits, Allocator> const &s)
auto is_any_of(char const *chars)

struct is_space
```

Public Functions

```
bool operator() (int c) const

namespace hpx
```

```
namespace string_util
```

Enums

```
enum token_compress_mode
```

Values:

off

on

Functions

```
template<typename Container, typename Predicate, typename CharT, typename Traits, typename Allocator>
void split (Container &container, std::basic_string<CharT, Traits, Allocator> const &str, Predicate &&pred, token_compress_mode compress_mode = token_compress_mode::off)
```

```
template<typename Container, typename Predicate>
```

```
void split (Container &container, char const *str, Predicate &&pred, token_compress_mode
compress_mode = token_compress_mode::off)
```

```
namespace hpx
```

```
namespace string_util
```

Functions

```
template<typename CharT, class Traits, class Alloc>
void trim (std::basic_string<CharT, Traits, Alloc> &s)
```

```
template<typename CharT, class Traits, class Alloc>
```

```
std::basic_string<CharT, Traits, Alloc> trim_copy (std::basic_string<CharT, Traits, Alloc>
const &s)
```

synchronization

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace experimental
```

```
template<typename ReadWriteT, typename ReadT, typename Allocator>
```

```
class async_rw_mutex
```

```
#include <async_rw_mutex.hpp> Read-write mutex where access is granted to a value through
senders.
```

The wrapped value is accessed through read and readwrite, both of which return senders which call set_value on a connected receiver when the wrapped value is safe to read or write. The senders send the value through a wrapper type which is implicitly convertible to a reference of the wrapped value. Read-only senders send wrappers that are convertible to const references.

A read-write sender gives exclusive access to the wrapped value, while a read-only sender gives shared (with other read-only senders) access to the value.

A void mutex acts as a mutex around some user-managed resource, i.e. the void mutex does not manage any value and the types sent by the senders are not convertible. The sent types are copyable and release access to the protected resource when released.

The order in which senders call set_value is determined by the order in which the senders are retrieved from the mutex. Connecting and starting the senders is thread-safe.

Retrieving senders from the mutex is not thread-safe.

The mutex is movable and non-copyable.

Public Types

```
template<>
using read_type = std::decay_t<ReadT> const

template<>
using readwrite_type = std::decay_t<ReadWriteT>

template<>
using value_type = readwrite_type

template<>
using read_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::async_

template<>
using readwrite_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::

template<>
using allocator_type = Allocator
```

Public Functions

```
async_rw_mutex()

template<typename U, typename = std::enable_if_t<!std::is_same<std::decay_t<U>, async_rw_mutex>::value>>
async_rw_mutex(U &&u, allocator_type const &alloc = {})

async_rw_mutex(async_rw_mutex&&)

async_rw_mutex &operator=(async_rw_mutex&&)

async_rw_mutex(async_rw_mutex const&)

async_rw_mutex &operator=(async_rw_mutex const&)

sender<detail::async_rw_mutex_access_type::read> read()

sender<detail::async_rw_mutex_access_type::readwrite> readwrite()
```

Private Types

```
template<>
using shared_state_type = detail::async_rw_mutex_shared_state<value_type>

template<>
using shared_state_ptr_type = std::shared_ptr<shared_state_type>
```

Private Members

```
value_type value
allocator_type alloc
detail::async_rw_mutex_access_type prev_access = detail::async_rw_mutex_access_type::readwrite
shared_state_ptr_type prev_state
shared_state_ptr_type state

template<detail::async_rw_mutex_access_type AccessType>
struct sender
```

Public Types

```
template<>
template<>
using access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, AccessType>
template<>
template<template<typename...> class Tuple, template<typename...> class Variantusing value_types = Variant<Tuple<access_type>>
template<>
template<template<typename...> class Variantusing error_types = Variant<std::exception_ptr>
```

Public Members

```
template<>
shared_state_ptr_type prev_state
template<>
shared_state_ptr_type state
```

Public Static Attributes

```
template<>
constexpr bool sends_done = false
```

Friends

```
template<typename R>
auto tag_invoke(hpx::execution::experimental::connect_t, sender &&s, R &&r)
```

```
template<typename R>
struct operation_state
```

Public Functions

```
template<>
template<typename R_>
operation_state(R_ &&r, shared_state_ptr_type prev_state, shared_state_ptr_type state)
```

```
template<>
template<>
operation_state(operation_state&&)
```

```
template<>
template<>
operation_state &operator=(operation_state&&)
```

```
template<>
template<>
operation_state(operation_state const&)
```

```
template<>
template<>
operation_state &operator=(operation_state const&)
```

Public Members

```
template<>
template<>
std::decay_t<R> r
```

```
template<>
template<>
shared_state_ptr_type prev_state
```

```
template<>
template<>
shared_state_ptr_type state
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)

template<typename Allocator>
class async_rw_mutex<void, void, Allocator>
```

Public Types

```
template<>
using read_type = void

template<>
using readwrite_type = void

template<>
using read_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::async_>

template<>
using readwrite_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::>

template<>
using allocator_type = Allocator
```

Public Functions

```
async_rw_mutex(allocator_type const &alloc = {})

async_rw_mutex(async_rw_mutex&&)

async_rw_mutex &operator=(async_rw_mutex&&)

async_rw_mutex(async_rw_mutex const&)

async_rw_mutex &operator=(async_rw_mutex const&)

sender<detail::async_rw_mutex_access_type::read> read()

sender<detail::async_rw_mutex_access_type::readwrite> readwrite()
```

Private Types

```
template<>
using shared_state_type = detail::async_rw_mutex_shared_state<void>

template<>
using shared_state_ptr_type = std::shared_ptr<shared_state_type>
```

Private Members

```
allocator_type alloc
detail::async_rwlock_access_type prev_access = detail::async_rwlock_access_type::readwrite
shared_state_ptr_type prev_state
shared_state_ptr_type state

template<detail::async_rwlock_access_type AccessType>
struct sender
```

Public Types

```
template<>
template<>
using access_type = detail::async_rwlock_access_wrapper<readwrite_type, read_type, AccessType>
template<>
template<template<typename...> class Tuple, template<typename...> class Variant>
using value_types = Variant<Tuple<access_type>>
template<>
template<template<typename...> class Variant>
using error_types = Variant<std::exception_ptr>
```

Public Members

```
template<>
shared_state_ptr_type prev_state
template<>
shared_state_ptr_type state
```

Public Static Attributes

```
template<>
constexpr bool sends_done = false
```

Friends

```
template<typename R>
auto tag_invoke(hpx::execution::experimental::connect_t, sender &&s, R &&r)
```

```
template<typename R>
struct operation_state
```

Public Functions

```
template<>
template<typename R_>
operation_state(R_ &&r, shared_state_ptr_type prev_state, shared_state_ptr_type
state)

template<>
template<>
operation_state(operation_state&&)

template<>
template<>
operation_state &operator=(operation_state&&)

template<>
template<>
operation_state(operation_state const&)

template<>
template<>
operation_state &operator=(operation_state const&)
```

Public Members

```
template<>
template<>
std::decay_t<R> r

template<>
template<>
shared_state_ptr_type prev_state

template<>
template<>
shared_state_ptr_type state
```

Friends

```
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)

namespace hpx

namespace lcos

namespace local

class barrier
#include <barrier.hpp> A barrier can be used to synchronize a specific number of threads, blocking all of the entering threads until all of the threads have entered the barrier.
```

Note A *barrier* is not a LCO in the sense that it has no global id and it can't be triggered using the action (parcel) mechanism. It is just a low level synchronization primitive allowing to synchronize a given number of *threads*.

Public Functions

barrier (*std::size_t number_of_threads*)

~barrier()

void wait ()

The function *wait* will block the number of entering *threads* (as given by the constructor parameter *number_of_threads*), releasing all waiting threads as soon as the last *thread* entered this function.

void count_up ()

The function *count_up* will increase the number of *threads* to be waited in *wait* function.

void reset (*std::size_t number_of_threads*)

The function *reset* will reset the number of *threads* as given by the function parameter *number_of_threads*. the newer coming *threads* executing the function *wait* will be waiting until *total_* is equal to *barrier_flag*. The last *thread* exiting the *wait* function will notify the newer *threads* waiting and the newer *threads* will get the reset *number_of_threads_*. The function *reset* can be executed while previous *threads* executing waiting after they have been waken up. Thus *total_* can not be reset to *barrier_flag* which will break the comparison condition under the function *wait*.

Private Types

typedef lcos::local::spinlock mutex_type

Private Members

std::size_t number_of_threads_

std::size_t total_

mutex_type mtx_

local::detail::condition_variable cond_

Private Static Attributes

constexpr std::size_t barrier_flag = static_cast<std::size_t>(1) << (CHAR_BIT * sizeof(std::size_t) - 1)

```
template<typename OnCompletion = detail::empty_oncompletion>
class cpp20_barrier
```

Public Types

```
template<>
using arrival_token = bool
```

Public Functions

```
HPX_NON_COPYABLE (cpp20_barrier)

cpp20_barrier (std::ptrdiff_t expected, OnCompletion completion = OnCompletion())
    HPX_NODISCARD arrival_token hpx::lcos::local::cpp20_barrier::arrive(std::ptrdiff_t
void wait (arrival_token &&old_phase) const
void arrive_and_wait ()
    Effects: Equivalent to: wait(arrive()).

void arrive_and_drop ()
```

Public Static Functions

```
static constexpr std::ptrdiff_t() hpx::lcos::local::cpp20_barrier::max()
```

Private Types

```
template<>
using mutex_type = lcos::local::spinlock
```

Private Functions

```
HPX_NODISCARD arrival_token hpx::lcos::local::cpp20_barrier::arrive_locked(std::ptrdiff_t
    expected, OnCompletion completion)
```

Private Members

```
mutex_type mtx_
local::detail::condition_variable cond_
std::ptrdiff_t expected_
std::ptrdiff_t arrived_
OnCompletion completion_
bool phase_

namespace hpx

namespace lcos

namespace local
```

TypeDefs

```
template<typename T>
using channel_mpmc = bounded_channel<T, hpx::lcos::local::spinlock>

template<typename T, typename Mutex = util::spinlock>
class bounded_channel
```

Public Functions

```
bounded_channel (std::size_t size)
bounded_channel (bounded_channel &&rhs)
bounded_channel &operator= (bounded_channel &&rhs)
~bounded_channel ()
bool get (T *val = nullptr) const
bool set (T &&t)
std::size_t close ()
std::size_t capacity () const
```

Protected Functions

```
std::size_t close (std::unique_lock<mutex_type> &l)
```

Private Types

```
template<>
using mutex_type = Mutex
```

Private Functions

```
bool is_full (std::size_t tail) const
bool is_empty (std::size_t head) const
```

Private Members

```
hpx::util::cache_aligned_data<mutex_type> mtx_
hpx::util::cache_aligned_data<std::size_t> head_
hpx::util::cache_aligned_data<std::size_t> tail_
std::size_t size_
std::unique_ptr<T[]> buffer_
bool closed_
```

```
namespace hpx
```

```
    namespace lcos
```

```
        namespace local
```

Typedefs

```
template<typename T>
using channel_mpsc = base_channel_mpsc<T, hpx::lcos::local::spinlock>
```

```
template<typename T, typename Mutex = util::spinlock>
class base_channel_mpsc
```

Public Functions

```
base_channel_mpsc (std::size_t size)
```

```
base_channel_mpsc (base_channel_mpsc &&rhs)
```

```
base_channel_mpsc &operator= (base_channel_mpsc &&rhs)
```

```
~base_channel_mpsc ()
```

```
bool get (T *val = nullptr) const
```

```
bool set (T &&t)
```

```
std::size_t close ()
```

```
std::size_t capacity () const
```

Private Types

```
template<>
using mutex_type = Mutex
```

Private Functions

```
bool is_full (std::size_t tail) const
```

```
bool is_empty (std::size_t head) const
```

Private Members

```
hpx::util::cache_aligned_data<std::atomic<std::size_t>> head_
hpx::util::cache_aligned_data<tail_data> tail_
std::size_t size_
std::unique_ptr<T[]> buffer_
std::atomic<bool> closed_

struct tail_data
```

Public Members

```
template<>
mutex_type mtx_
template<>
std::atomic<std::size_t> tail_
namespace hpx

namespace lcos

namespace local

template<typename T>
class channel_spse
```

Public Functions

```
channel_spse (std::size_t size)
channel_spse (channel_spse &&rhs)
channel_spse &operator= (channel_spse &&rhs)
~channel_spse ()
bool get (T *val = nullptr) const
bool set (T &&t)
std::size_t close ()
std::size_t capacity () const
```

Private Functions

```
bool is_full (std::size_t tail) const
bool is_empty (std::size_t head) const
```

Private Members

```
hpx::util::cache_aligned_data<std::atomic<std::size_t>> head_
hpx::util::cache_aligned_data<std::atomic<std::size_t>> tail_
std::size_t size_
std::unique_ptr<T[]> buffer_
std::atomic<bool> closed_
```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

Enums

```
enum cv_status
  Values:
    no_timeout
    timeout
    error

class condition_variable
```

Public Functions

```
condition_variable ()
~condition_variable ()

void notify_one (error_code &ec = throws)
void notify_all (error_code &ec = throws)

template<typename Mutex>
void wait (std::unique_lock<Mutex> &lock, error_code &ec = throws)

template<typename Mutex, typename Predicate>
void wait (std::unique_lock<Mutex> &lock, Predicate pred, error_code& = throws)

template<typename Mutex>
cv_status wait_until (std::unique_lock<Mutex> &lock, hpx::chrono::steady_time_point
                      const &abs_time, error_code &ec = throws)
```

```

template<typename Mutex, typename Predicate>
bool wait_until(std::unique_lock<Mutex> &lock, hpx::chrono::steady_time_point
                 const &abs_time, Predicate pred, error_code &ec = throws)

template<typename Mutex>
cv_status wait_for(std::unique_lock<Mutex> &lock, hpx::chrono::steady_duration
                    const &rel_time, error_code &ec = throws)

template<typename Mutex, typename Predicate>
bool wait_for(std::unique_lock<Mutex> &lock, hpx::chrono::steady_duration const
               &rel_time, Predicate pred, error_code &ec = throws)

```

Private Types

```

using mutex_type = detail::condition_variable_data::mutex_type
using data_type = hpx::memory::intrusive_ptr<detail::condition_variable_data>

```

Private Members

```

hpx::util::cache_aligned_data_derived<data_type> data_
class condition_variable_any

```

Public Functions

```

condition_variable_any()

~condition_variable_any()

void notify_one(error_code &ec = throws)

void notify_all(error_code &ec = throws)

template<typename Lock>
void wait(Lock &lock, error_code &ec = throws)

template<typename Lock, typename Predicate>
void wait(Lock &lock, Predicate pred, error_code &ec = throws)

template<typename Lock>
cv_status wait_until(Lock &lock, hpx::chrono::steady_time_point const &abs_time, error_code &ec = throws)

template<typename Lock, typename Predicate>
bool wait_until(Lock &lock, hpx::chrono::steady_time_point const &abs_time, Predicate pred, error_code &ec = throws)

template<typename Lock>
cv_status wait_for(Lock &lock, hpx::chrono::steady_duration const &rel_time, error_code &ec = throws)

template<typename Lock, typename Predicate>
bool wait_for(Lock &lock, hpx::chrono::steady_duration const &rel_time, Predicate pred, error_code &ec = throws)

template<typename Lock, typename Predicate>

```

```
bool wait (Lock &lock, stop_token stoken, Predicate pred, error_code &ec = throws)
template<typename Lock, typename Predicate>
bool wait_until (Lock &lock, stop_token stoken, hpx::chrono::steady_time_point const &abs_time, Predicate pred, error_code &ec = throws)
template<typename Lock, typename Predicate>
bool wait_for (Lock &lock, stop_token stoken, hpx::chrono::steady_duration const &rel_time, Predicate pred, error_code &ec = throws)
```

Private Types

```
using mutex_type = detail::condition_variable_data::mutex_type
using data_type = hpx::memory::intrusive_ptr<detail::condition_variable_data>
```

Private Members

```
hpx::util::cache_aligned_data_derived<data_type> data_
```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

Typedefs

```
typedef counting_semaphore_var counting_semaphore
```

```
template<typename Mutex = hpx::lcos::local::spinlock, int N = 0>
```

```
class counting_semaphore_var : private hpx::lcos::local::cpp20_counting_semaphore<PTRDIFF_MAX,
```

Public Functions

```
counting_semaphore_var (std::ptrdiff_t value = N)
```

```
counting_semaphore_var (counting_semaphore_var const&)
```

```
counting_semaphore_var &operator= (counting_semaphore_var const&)
```

```
void wait (std::ptrdiff_t count = 1)
```

```
bool try_wait (std::ptrdiff_t count = 1)
```

```
void signal (std::ptrdiff_t count = 1)
```

```
Signal the semaphore.
```

```
std::ptrdiff_t signal_all ()
```

Private Types

```
template<>
using mutex_type = Mutex

template<typename Mutex = hpx::lcos::local::spinlock>
class cpp20_binary_semaphore : public hpx::lcos::local::cpp20_counting_semaphore<1, hpx::lcos::local::
```

Public Functions

```
HPX_NON_COPYABLE (cpp20_binary_semaphore)

cpp20_binary_semaphore (std::ptrdiff_t value = 1)

~cpp20_binary_semaphore ()

template<std::ptrdiff_t Least.MaxValue = PTRDIFF_MAX, typename Mutex = hpx::lcos::local::spinlock>
class cpp20_counting_semaphore
```

Public Functions

```
HPX_NON_COPYABLE (cpp20_counting_semaphore)

cpp20_counting_semaphore (std::ptrdiff_t value)

~cpp20_counting_semaphore ()

void release (std::ptrdiff_t update = 1)

bool try_acquire ()

void acquire ()

bool try_acquire_until (hpx::chrono::steady_time_point const &abs_time)

bool try_acquire_for (hpx::chrono::steady_duration const &rel_time)
```

Public Static Functions

```
static constexpr std::ptrdiff_t () hpx::lcos::local::cpp20_counting_semaphore::
```

Protected Types

```
template<>
using mutex_type = Mutex
```

Protected Attributes

```
mutex_type mtx_
detail::counting_semaphore sem_

namespace hpx

namespace lcos

namespace local

class event
#include <event.hpp> Event semaphores can be used for synchronizing multiple threads that need
to wait for an event to occur. When the event occurs, all threads waiting for the event are woken
up.
```

Public Functions

```
event ()
Construct a new event semaphore.

bool occurred ()
Check if the event has occurred.

void wait ()
Wait for the event to occur.

void set ()
Release all threads waiting on this semaphore.

void reset ()
Reset the event.
```

Private Types

```
typedef lcos::local::spinlock mutex_type
```

Private Functions

```
void wait_locked (std::unique_lock<mutex_type> &l)
void set_locked (std::unique_lock<mutex_type> l)
```

Private Members

```

mutex_type mtx_
    This mutex protects the queue.

local::detail::condition_variable cond_

std::atomic<bool> event_

namespace hpx

namespace lcos

namespace local

class cpp20_latch
#include <latch.hpp> Latches are a thread coordination mechanism that allow one or more threads to block until an operation is completed. An individual latch is a singleuse object; once the operation has been completed, the latch cannot be reused.

Subclassed by hpx::lcos::local::latch

```

Public Functions

```

HPX_NON_COPYABLE (cpp20_latch)

cpp20_latch (std::ptrdiff_t count)
    Initialize the latch

    Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

~cpp20_latch()
    Requires: No threads are blocked at the synchronization point.

```

Note May be called even if some threads have not yet returned from *wait()* or *count_down_and_wait()*, provided that *counter_* is 0.

Note The destructor might not return until all threads have exited *wait()* or *count_down_and_wait()*.

Note It is the caller's responsibility to ensure that no other thread enters *wait()* after one thread has called the destructor. This may require additional coordination.

```

void count_down (std::ptrdiff_t update)
    Decrements counter_ by n. Does not block.

    Requires: counter_ >= n and n >= 0.

    Synchronization: Synchronizes with all calls that block on this latch and with all try_wait calls on this latch that return true .

```

Exceptions

- Nothing.:

```

bool try_wait() const
    Returns: With very low probability false. Otherwise counter == 0.

```

```
void wait() const
```

If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization point until counter_ reaches 0.

Exceptions

- Nothing..:

```
void arrive_and_wait (std::ptrdiff_t update = 1)
```

Effects: Equivalent to: count_down(update); wait();

Public Static Functions

```
static constexpr std::ptrdiff_t() hpx::lcos::local::cpp20_latch::max()
```

Returns: The maximum value of counter that the implementation supports.

Protected Types

```
using mutex_type = lcos::local::spinlock
```

Protected Attributes

```
util::cache_line_data<mutex_type> mtx_
```

```
util::cache_line_data<local::detail::condition_variable> cond_
```

```
std::atomic<std::ptrdiff_t> counter_
```

```
bool notified_
```

```
class latch : public hpx::lcos::local::cpp20_latch
```

#include <latch.hpp> A latch maintains an internal counter_ that is initialized when the latch is created. Threads may block at a synchronization point waiting for counter_ to be decremented to 0. When counter_ reaches 0, all such blocked threads are released.

Calls to *countdown_and_wait()* , *count_down()* , *wait()* , *is_ready()*, *count_up()* , and *reset()* behave as atomic operations.

Note A local::latch is not an LCO in the sense that it has no global id and it can't be triggered using the action (parcel) mechanism. Use lcos::latch instead if this is required. It is just a low level synchronization primitive allowing to synchronize a given number of *threads*.

Public Functions

```
HPX_NON_COPYABLE (latch)
```

```
latch (std::ptrdiff_t count)
```

Initialize the latch

Requires: count ≥ 0 . Synchronization: None Postconditions: counter_ == count.

```
~latch ()
```

Requires: No threads are blocked at the synchronization point.

Note May be called even if some threads have not yet returned from `wait()` or `count_down_and_wait()`, provided that `counter_` is 0.

Note The destructor might not return until all threads have exited `wait()` or `count_down_and_wait()`.

Note It is the caller's responsibility to ensure that no other thread enters `wait()` after one thread has called the destructor. This may require additional coordination.

`void count_down_and_wait ()`

Decrements `counter_` by 1. Blocks at the synchronization point until `counter_` reaches 0.

Requires: `counter_ > 0`.

Synchronization: Synchronizes with all calls that block on this latch and with all `is_ready` calls on this latch that return true.

Exceptions

- Nothing.:

`bool is_ready () const`

Returns: `counter_ == 0`. Does not block.

Exceptions

- Nothing.:

`void abort_all ()`

`void count_up (std::ptrdiff_t n)`

Increments `counter_` by `n`. Does not block.

Requires: `n >= 0`.

Exceptions

- Nothing.:

`void reset (std::ptrdiff_t n)`

Reset `counter_` to `n`. Does not block.

Requires: `n >= 0`.

Exceptions

- Nothing.:

`bool reset_if_needed_and_count_up (std::ptrdiff_t n, std::ptrdiff_t count)`

Effects: Equivalent to: if (`is_ready()`) `reset(count)`; `count_up(n)`; Returns: true if the latch was reset

`namespace hpx`

`namespace lcos`

`namespace local`

Functions

```
template<typename Mutex>
void swap(upgrade_lock<Mutex> &lhs, upgrade_lock<Mutex> &rhs)

template<typename Mutex>
class upgrade_lock
```

Public Types

```
template<>
using mutex_type = Mutex
```

Public Functions

```
upgrade_lock(upgrade_lock const&)

upgrade_lock &operator=(upgrade_lock const&)

upgrade_lock()

upgrade_lock(Mutex &m_)

upgrade_lock(Mutex &m_, std::adopt_lock_t)

upgrade_lock(Mutex &m_, std::defer_lock_t)

upgrade_lock(Mutex &m_, std::try_to_lock_t)

upgrade_lock(upgrade_lock<Mutex> &&other)

upgrade_lock(std::unique_lock<Mutex> &&other)

upgrade_lock &operator=(upgrade_lock<Mutex> &&other)

void swap(upgrade_lock &other)

Mutex *mutex() const

Mutex *release()

~upgrade_lock()

void lock()

bool try_lock()

void unlock()

operator bool() const

bool owns_lock() const
```

Protected Attributes

```
Mutex *m
bool is_locked
```

Friends

```
friend hpx::lcos::local::upgrade_to_unique_lock
template<typename Mutex>
class upgrade_to_unique_lock
```

Public Types

```
template<>
using mutex_type = Mutex
```

Public Functions

```
upgrade_to_unique_lock(upgrade_to_unique_lock const&)
upgrade_to_unique_lock &operator=(upgrade_to_unique_lock const&)
upgrade_to_unique_lock(upgrade_lock<Mutex> &m_)
~upgrade_to_unique_lock()
upgrade_to_unique_lock(upgrade_to_unique_lock<Mutex> &&other)
upgrade_to_unique_lock &operator=(upgrade_to_unique_lock<Mutex> &&other)
void swap(upgrade_to_unique_lock &other)
operator bool() const
bool owns_lock() const
Mutex *mutex() const
```

Private Members

```
upgrade_lock<Mutex> *source
std::unique_lock<Mutex> exclusive
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```
class mutex
Subclassed by hpx::lcos::local::timed_mutex
```

Public Functions

```
HPX_NON_COPYABLE (mutex)

mutex (char const *const description = "")

~mutex ()

void lock (char const *description, error_code &ec = throws)

void lock (error_code &ec = throws)

bool try_lock (char const *description, error_code &ec = throws)

bool try_lock (error_code &ec = throws)

void unlock (error_code &ec = throws)
```

Protected Types

```
typedef lcos::local::spinlock mutex_type
```

Protected Attributes

```
mutex_type mtx_
threads::thread_id_type owner_id_
lcos::local::detail::condition_variable cond_

class timed_mutex : private hpx::lcos::local::mutex
```

Public Functions

```
HPX_NON_COPYABLE (timed_mutex)

timed_mutex (char const *const description = "")

~timed_mutex ()

bool try_lock_until (hpx::chrono::steady_time_point const &abs_time, char const
                     *description, error_code &ec = throws)

bool try_lock_until (hpx::chrono::steady_time_point const &abs_time, error_code
                     &ec = throws)

bool try_lock_for (hpx::chrono::steady_duration const &rel_time, char const *de-
                     scription, error_code &ec = throws)

bool try_lock_for (hpx::chrono::steady_duration const &rel_time, error_code &ec =
                     throws)

void lock (char const *description, error_code &ec = throws)

void lock (error_code &ec = throws)

bool try_lock (char const *description, error_code &ec = throws)
```

```
bool try_lock(error_code &ec = throws)
void unlock(error_code &ec = throws)

namespace threads
```

Typedefs

```
using thread_id_ref_type = thread_id_ref
using thread_self = coroutines::detail::coroutine_self
```

Functions

```
thread_id get_self_id()
```

The function *get_self_id* returns the HPX thread id of the current thread (or zero if the current thread is not a HPX thread).

```
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.

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```
struct no_mutex
```

Public Functions

```
void lock()
```

```
bool try_lock()
```

```
void unlock()
```

Defines

```
HPX_ONCE_INIT
```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

Functions

```
template<typename F, typename ...Args>
void call_once (once_flag &flag, F &&f, Args&&... args)

struct once_flag
```

Public Functions

```
HPX_NON_COPYABLE (once_flag)

once_flag ()
```

Private Members

```
std::atomic<long> status_
lcos::local::event event_
```

Friends

```
template<typename F, typename ...Args>
void call_once (once_flag &flag, F &&f, Args&&... args)

namespace hpx

namespace lcos

namespace local
```

Typedefs

```
using recursive_mutex = detail::recursive_mutex_impl<>

namespace hpx

namespace lcos

namespace local
```

TypeDefs

```
typedef detail::shared_mutex shared_mutex

namespace hpx

namespace lcos

namespace local
```

TypeDefs

```
typedef sliding_semaphore_var sliding_semaphore
```

```
template<typename Mutex = hpx::lcos::local::spinlock>
class sliding_semaphore_var
```

#include <sliding_semaphore.hpp> A semaphore is a protected variable (an entity storing a value) or abstract data type (an entity grouping several variables that may or may not be numerical) which constitutes the classic method for restricting access to shared resources, such as shared memory, in a multiprogramming environment. Semaphores exist in many variants, though usually the term refers to a counting semaphore, since a binary semaphore is better known as a mutex. A counting semaphore is a counter for a set of available resources, rather than a locked/unlocked flag of a single resource. It was invented by Edsger Dijkstra. Semaphores are the classic solution to preventing race conditions in the dining philosophers problem, although they do not prevent resource deadlocks.

Sliding semaphores can be used for synchronizing multiple threads as well: one thread waiting for several other threads to touch (signal) the semaphore, or several threads waiting for one other thread to touch this semaphore. The difference to a counting semaphore is that a sliding semaphore will not limit the number of threads which are allowed to proceed, but will make sure that the difference between the (arbitrary) number passed to set and wait does not exceed a given threshold.

Public Functions

```
sliding_semaphore_var (std::int64_t max_difference, std::int64_t lower_limit = 0)
```

Construct a new sliding semaphore.

Parameters

- **max_difference**: [in] The max difference between the upper limit (as set by *wait()*) and the lower limit (as set by *signal()*) which is allowed without suspending any thread calling *wait()*.
- **lower_limit**: [in] The initial lower limit.

```
void set_max_difference (std::int64_t max_difference, std::int64_t lower_limit = 0)
```

Set/Change the difference that will cause the semaphore to trigger.

Parameters

- **max_difference**: [in] The max difference between the upper limit (as set by *wait()*) and the lower limit (as set by *signal()*) which is allowed without suspending any thread calling *wait()*.

- `lower_limit`: [in] The initial lower limit.

```
void wait (std::int64_t upper_limit)  
    Wait for the semaphore to be signaled.
```

Parameters

- `upper_limit`: [in] The new upper limit. The calling thread will be suspended if the difference between this value and the largest `lower_limit` which was set by `signal()` is larger than the `max_difference`.

```
bool try_wait (std::int64_t upper_limit = 1)  
    Try to wait for the semaphore to be signaled.
```

Return The function returns true if the calling thread would not block if it was calling `wait()`.

Parameters

- `upper_limit`: [in] The new upper limit. The calling thread will be suspended if the difference between this value and the largest `lower_limit` which was set by `signal()` is larger than the `max_difference`.

```
void signal (std::int64_t lower_limit)  
    Signal the semaphore.
```

Parameters

- `lower_limit`: [in] The new lower limit. This will update the current lower limit of this semaphore. It will also re-schedule all suspended threads for which their associated upper limit is not larger than the lower limit plus the `max_difference`.

```
std::int64_t signal_all ()
```

Private Types

```
typedef Mutex mutex_type
```

Private Members

```
mutex_type mtx_  
detail::sliding_semaphore sem_
```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```
struct spinlock
```

Public Functions

```
HPX_NON_COPYABLE (spinlock)
spinlock (char const *const desc = "hpx::lcos::local::spinlock")
~spinlock ()
void lock ()
bool try_lock ()
void unlock ()
```

Private Functions

```
bool acquire_lock ()
void relinquish_lock ()
bool is_locked () const
```

Private Members

```
std::atomic<bool> v_
namespace hpx
namespace lcos
namespace local

struct spinlock_no_backoff
#include <spinlock_no_backoff.hpp> boost::mutex-compatible spinlock class
```

Public Functions

```
HPX_NON_COPYABLE (spinlock_no_backoff)
spinlock_no_backoff ()
~spinlock_no_backoff ()
void lock ()
bool try_lock ()
void unlock ()
```

Private Functions

```
bool acquire_lock ()  
void relinquish_lock ()  
bool is_locked () const
```

Private Members

```
std::atomic<bool> v_  
namespace hpx  
  
namespace lcos  
  
namespace local  
  
template<typename Tag, std::size_t N = HPX_HAVE_SPINLOCK_POOL_NUM>  
class spinlock_pool
```

Public Static Functions

```
static lcos::local::spinlock &spinlock_for (void const *pv)
```

Private Static Attributes

```
util::cache_aligned_data<lcos::local::spinlock> pool_  
class scoped_lock
```

Public Functions

```
template<>  
HPX_NON_COPYABLE (scoped_lock)  
  
template<>  
scoped_lock (void const *pv)  
  
template<>  
~scoped_lock ()  
  
template<>  
void lock ()  
  
template<>  
void unlock ()
```

Private Members

```
template<>
hpx::lcos::local::spinlock &sp_
namespace hpx
```

Functions

```
template<typename Callback>
stop_callback<typename std::decay<Callback>::type> make_stop_callback(stop_token const
&st,           Callback
&&cb)

template<typename Callback>
stop_callback<typename std::decay<Callback>::type> make_stop_callback(stop_token &&st,
Callback &&cb)

template<typename Callback>
stop_callback(stop_token, Callback)

void swap(stop_token &lhs, stop_token &rhs)
void swap(stop_source &lhs, stop_source &rhs)
```

Variables

```
constexpr nostopstate_t nostopstate = {}

struct nostopstate_t
```

Public Functions

```
nostopstate_t()

template<typename Callback>
class stop_callback : private hpx::detail::stop_callback_base
```

Public Types

```
template<>
using callback_type = Callback
```

Public Functions

```
template<typename CB, typename Enable = typename std::enable_if<std::is_constructible<Callback, CB>::value>::type>
stop_callback(stop_token const &st, CB &&cb)

template<typename CB, typename Enable = typename std::enable_if<std::is_constructible<Callback, CB>::value>::type>
stop_callback(stop_token &&st, CB &&cb)

~stop_callback()
```

```
stop_callback (stop_callback const&)
stop_callback (stop_callback&&)
stop_callback &operator= (stop_callback const&)
stop_callback &operator= (stop_callback&&)
```

Private Functions

```
void execute ()
```

Private Members

```
Callback callback_
hpx::memory::intrusive_ptr<detail::stop_state> state_
class stop_source
```

Public Functions

```
stop_source ()
stop_source (nostopstate_t)
stop_source (stop_source const &rhs)
stop_source (stop_source&&)
stop_source &operator= (stop_source const &rhs)
stop_source &operator= (stop_source&&)
~stop_source ()

void swap (stop_source &s)
HPX_NODISCARD stop_token hpx::stop_source::get_token() const
HPX_NODISCARD bool hpx::stop_source::stop_possible() const
HPX_NODISCARD bool hpx::stop_source::stop_requested() const
bool request_stop ()
```

Private Members

```
hpx::memory::intrusive_ptr<detail::stop_state> state_
```

Friends

```
HPX_NODISCARD friend bool operator==(stop_source const & lhs, stop_source const &
HPX_NODISCARD friend bool operator!=(stop_source const & lhs, stop_source const &

class stop_token
```

Public Functions

```
stop_token()
stop_token(stop_token const &rhs)
stop_token(stop_token&&)
stop_token &operator=(stop_token const &rhs)
stop_token &operator=(stop_token&&)
~stop_token()
void swap(stop_token &s)
HPX_NODISCARD bool hpx::stop_token::stop_requested() const
HPX_NODISCARD bool hpx::stop_token::stop_possible() const
```

Private Functions

```
stop_token(hpx::memory::intrusive_ptr<detail::stop_state> const &state)
```

Private Members

```
hpx::memory::intrusive_ptr<detail::stop_state> state_
```

Friends

```
friend hpx::stop_callback
friend hpx::stop_source
HPX_NODISCARD friend bool operator==(stop_token const & lhs, stop_token const &
HPX_NODISCARD friend bool operator!=(stop_token const & lhs, stop_token const &
```

tag_invoke

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
    namespace util
```

Typedefs

```
template<typename F, typename ...Ts>
using invoke_result_t = typename invoke_result<F, Ts...>::type
```

```
namespace hpx
```

```
    namespace functional
```

Typedefs

```
template<typename Tag, typename ...Args>
using tag_invoke_result = invoke_result<decltype(tag_invoke), Tag, Args...>
    hpx::functional::tag_invoke_result<Tag, Args...> is the trait returning the result type of the call hpx::functional::tag_invoke. This can be used in a SFINAE context.
```

```
template<typename Tag, typename ...Args>
using tag_invoke_result_t = typename tag_invoke_result<Tag, Args...>::type
    hpx::functional::tag_invoke_result_t<Tag, Args...> evaluates to
    hpx::functional::tag_invoke_result_t<Tag, Args...>::type
```

Variables

```
constexpr unspecified tag_invoke = unspecified
```

The `hpx::functional::tag_invoke` name defines a `constexpr` object that is invocable with one or more arguments. The first argument is a ‘tag’ (typically a DPO). It is only invocable if an overload of `tag_invoke()` that accepts the same arguments could be found via ADL.

The evaluation of the expression `hpx::tag_invoke(tag, args...)` is equivalent to evaluating the unqualified call to `tag_invoke(decay-copy(tag), HPX_FORWARD(Args, args)...)`.

`hpx::functional::tag_invoke` is implemented against P1895.

Example: Defining a new customization point `foo`:

```
namespace mylib {
    inline constexpr
        struct foo_fn final : hpx::functional::tag<foo_fn>
    {
        } foo{};
}
```

Defining an object `bar` which customizes `foo`:

```
struct bar
{
    int x = 42;

    friend constexpr int tag_invoke(mylib::foo_fn, bar const& x)
    {
        return b.x;
    }
};
```

Using the customization point:

```
static_assert(42 == mylib::foo(bar{}), "The answer is 42");
```

```
template<typename Tag, typename ...Args>
constexpr bool is_tag_invocable_v = is_tag_invocable<Tag, Args...>::value
    hpx::functional::is_tag_invocable_v<Tag, Args...> evaluates to
    hpx::functional::is_tag_invocable<Tag, Args...>::value

template<typename Tag, typename ...Args>
constexpr bool is_nothrow_tag_invocable_v = is_nothrow_tag_invocable<Tag, Args...>::value
    hpx::functional::is_tag_invocable_v<Tag, Args...> evaluates to
    hpx::functional::is_tag_invocable<Tag, Args...>::value

template<typename Tag, typename ...Args>
struct is_nothrow_tag_invocable
    #include <tag_invoke.hpp> hpx::functional::is_nothrow_tag_invocable<Tag,
    Args...> is std::true_type if an overload of tag_invoke(tag, args...) can be found via
    ADL and is noexcept.

template<typename Tag, typename ...Args>
struct is_tag_invocable
    #include <tag_invoke.hpp> hpx::functional::is_tag_invocable<Tag, Args...> is
    std::true_type if an overload of tag_invoke(tag, args...) can be found via ADL.

template<typename Tag>
struct tag
    #include <tag_invoke.hpp> hpx::functional::tag<Tag> defines a base class that implements
    the necessary tag dispatching functionality for a given type Tag

template<typename Tag>
struct tag_noexcept
    #include <tag_invoke.hpp> hpx::functional::tag_noexcept<Tag> defines a base class
    that implements the necessary tag dispatching functionality for a given type Tag. The implementation
    has to be noexcept

namespace hpx
```

Variables

```
template<typename F, typename ...Ts>
constexpr bool is_invocable_v = is_invocable<F, Ts...>::value

template<typename R, typename F, typename ...Ts>
constexpr bool is_invocable_r_v = is_invocable_r<R, F, Ts...>::value

namespace traits
```

TypeDefs

```
typedef hpx::is_invocable_r<R, F, Ts...> instead
```

testing

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_TEST (...)

HPX_TEST_(...)
HPX_TEST_1 (expr)
HPX_TEST_2 (strm, expr)
HPX_TEST_IMPL (fixture, expr)
HPX_TEST_MSG (...)

HPX_TEST_MSG_(...)
HPX_TEST_MSG_2 (expr, msg)
HPX_TEST_MSG_3 (strm, expr, msg)
HPX_TEST_MSG_IMPL (fixture, expr, msg)
HPX_TEST_EQ (...)

HPX_TEST_EQ_(...)
HPX_TEST_EQ_2 (expr1, expr2)
HPX_TEST_EQ_3 (strm, expr1, expr2)
HPX_TEST_EQ_IMPL (fixture, expr1, expr2)
HPX_TEST_NEQ (...)

HPX_TEST_NEQ_(...)
HPX_TEST_NEQ_2 (expr1, expr2)
HPX_TEST_NEQ_3 (strm, expr1, expr2)
HPX_TEST_NEQ_IMPL (fixture, expr1, expr2)
HPX_TEST_LT (...)

HPX_TEST_LT_(...)
```

```
HPX_TEST_LT_2 (expr1, expr2)
HPX_TEST_LT_3 (strm, expr1, expr2)
HPX_TEST_LT_IMPL (fixture, expr1, expr2)
HPX_TEST_LTE (...)
HPX_TEST_LTE_ (...)
HPX_TEST_LTE_2 (expr1, expr2)
HPX_TEST_LTE_3 (strm, expr1, expr2)
HPX_TEST_LTE_IMPL (fixture, expr1, expr2)
HPX_TEST_RANGE (...)
HPX_TEST_RANGE_ (...)
HPX_TEST_RANGE_3 (expr1, expr2, expr3)
HPX_TEST_RANGE_4 (strm, expr1, expr2, expr3)
HPX_TEST_RANGE_IMPL (fixture, expr1, expr2, expr3)
HPX_TEST_EQ_MSG (...)
HPX_TEST_EQ_MSG_ (...)
HPX_TEST_EQ_MSG_3 (expr1, expr2, msg)
HPX_TEST_EQ_MSG_4 (strm, expr1, expr2, msg)
HPX_TEST_EQ_MSG_IMPL (fixture, expr1, expr2, msg)
HPX_TEST_NEQ_MSG (...)
HPX_TEST_NEQ_MSG_ (...)
HPX_TEST_NEQ_MSG_3 (expr1, expr2, msg)
HPX_TEST_NEQ_MSG_4 (strm, expr1, expr2, msg)
HPX_TEST_NEQ_MSG_IMPL (fixture, expr1, expr2, msg)
HPX_TEST_LT_MSG (...)
HPX_TEST_LT_MSG_ (...)
HPX_TEST_LT_MSG_3 (expr1, expr2, msg)
HPX_TEST_LT_MSG_4 (strm, expr1, expr2, msg)
HPX_TEST_LT_MSG_IMPL (fixture, expr1, expr2, msg)
HPX_TEST_LTE_MSG (...)
HPX_TEST_LTE_MSG_ (...)
HPX_TEST_LTE_MSG_3 (expr1, expr2, msg)
HPX_TEST_LTE_MSG_4 (strm, expr1, expr2, msg)
HPX_TEST_LTE_MSG_IMPL (fixture, expr1, expr2, msg)
HPX_TEST_RANGE_MSG (...)
HPX_TEST_RANGE_MSG_ (...)
HPX_TEST_RANGE_MSG_4 (expr1, expr2, expr3, msg)
```

```
HPX_TEST_RANGE_MSG_5 (strm, expr1, expr2, expr3, msg)
HPX_TEST_RANGE_MSG_IMPL (fixture, expr1, expr2, expr3, msg)
HPX_SANITY (...)
HPX_SANITY_ (...)
HPX_SANITY_1 (expr)
HPX_SANITY_2 (strm, expr)
HPX_SANITY_IMPL (fixture, expr)
HPX_SANITY_MSG (...)
HPX_SANITY_MSG_ (...)
HPX_SANITY_MSG_2 (expr, msg)
HPX_SANITY_MSG_3 (strm, expr, msg)
HPX_SANITY_MSG_IMPL (fixture, expr, msg)
HPX_SANITY_EQ (...)
HPX_SANITY_EQ_ (...)
HPX_SANITY_EQ_2 (expr1, expr2)
HPX_SANITY_EQ_3 (strm, expr1, expr2)
HPX_SANITY_EQ_IMPL (fixture, expr1, expr2)
HPX_SANITY_NEQ (...)
HPX_SANITY_NEQ_ (...)
HPX_SANITY_NEQ_2 (expr1, expr2)
HPX_SANITY_NEQ_3 (strm, expr1, expr2)
HPX_SANITY_NEQ_IMPL (fixture, expr1, expr2)
HPX_SANITY_LT (...)
HPX_SANITY_LT_ (...)
HPX_SANITY_LT_2 (expr1, expr2)
HPX_SANITY_LT_3 (strm, expr1, expr2)
HPX_SANITY_LT_IMPL (fixture, expr1, expr2)
HPX_SANITY_LTE (...)
HPX_SANITY_LTE_ (...)
HPX_SANITY_LTE_2 (expr1, expr2)
HPX_SANITY_LTE_3 (strm, expr1, expr2)
HPX_SANITY_LTE_IMPL (fixture, expr1, expr2)
HPX_SANITY_RANGE (...)
HPX_SANITY_RANGE_ (...)
HPX_SANITY_RANGE_3 (expr1, expr2, expr3)
HPX_SANITY_RANGE_4 (strm, expr1, expr2, expr3)
```

```

HPX_SANITY_RANGE_IMPL(fixture, expr1, expr2, expr3)
HPX_SANITY_EQ_MSG(...)
HPX_SANITY_EQ_MSG_(...)
HPX_SANITY_EQ_MSG_3(expr1, expr2, msg)
HPX_SANITY_EQ_MSG_4(strm, expr1, expr2, msg)
HPX_SANITY_EQ_MSG_IMPL(fixture, expr1, expr2, msg)
HPX_TEST_THROW(...)
HPX_TEST_THROW_(...)
HPX_TEST_THROW_2(expression, exception)
HPX_TEST_THROW_3(strm, expression, exception)
HPX_TEST_THROW_IMPL(fixture, expression, exception)
namespace hpx

```

```
namespace util
```

Typedefs

```
using test_failure_handler_type=function_nonser<void ()>
```

Enums

```
enum counter_type
  Values:
    counter_sanity
    counter_test
```

Functions

```

void set_test_failure_handler(test_failure_handler_type f)
int report_errors()
int report_errors(std::ostream &stream)
void print_cdash_timing(const char *name, double time)
void print_cdash_timing(const char *name, std::uint64_t time)
namespace hpx

```



```
namespace util
```

Functions

```
void perftests_report (std::string const &name, std::string const &exec, const  
                      std::size_t steps, function_nonser<void> void  
> &&test  
  
void perftests_print_times ()
```

thread_pool_util

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace threads
```

Functions

```
hpx::future<void> resume_processing_unit (thread_pool_base &pool, std::size_t virt_core)  
Resumes the given processing unit. When the processing unit has been resumed the returned future  
will be ready.
```

Note Can only be called from an HPX thread. Use `resume_processing_unit_cb` or to resume the processing unit from outside HPX. Requires that the pool has `threads::policies::enable_elasticity` set.

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.

```
void resume_processing_unit_cb (thread_pool_base &pool,  
                               util::function_nonser<void> void  
> callback, std::size_t virt_core, error_code &ec = throwsResumes 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.

```
hpx::future<void> suspend_processing_unit (thread_pool_base &pool, std::size_t  
                                              virt_core)
```

Suspends the given processing unit. When the processing unit has been suspended the returned future will be ready.

Note Can only be called from an HPX thread. Use `suspend_processing_unit_cb` or to suspend the processing unit from outside HPX. Requires that the pool has `threads::policies::enable_elasticity` set.

Return A `future<void>` which is ready when the given processing unit has been suspended.

Parameters

- `virt_core`: [in] The processing unit on the pool to be suspended. The processing units are indexed starting from 0.

Exceptions

- `hpx::exception`: if called from outside the HPX runtime.

```
void suspend_processing_unit_cb(util::function_nonser<void> void
> callback, thread_pool_base &pool, std::size_t virt_core, error_code &ec = throwsSuspends 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.

```
hpx::future<void> resume_pool(thread_pool_base &pool)
```

Resumes the thread pool. When the all OS threads on the thread pool have been resumed the returned future will be ready.

Note Can only be called from an HPX thread. Use `resume_cb` or `resume_direct` to suspend the pool from outside HPX.

Return A `future<void>` which is ready when the thread pool has been resumed.

Exceptions

- `hpx::exception`: if called from outside the HPX runtime.

```
void resume_pool_cb(thread_pool_base &pool, util::function_nonser<void> void
> callback, error_code &ec = throwsResumes 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.

```
hpx::future<void> suspend_pool(thread_pool_base &pool)
```

Suspends the thread pool. When the all OS threads on the thread pool have been suspended the returned future will be ready.

Note Can only be called from an HPX thread. Use `suspend_cb` or `suspend_direct` to suspend the pool from outside HPX. A thread pool cannot be suspended from an HPX thread running on the pool itself.

Return A `future<void>` which is ready when the thread pool has been suspended.

Exceptions

- `hpx::exception`: if called from outside the HPX runtime.

```
void suspend_pool_cb(thread_pool_base &pool, util::function_nonser<void> void
> callback, error_code &ec = throwsSuspends the thread pool. Takes a callback as a parameter which
will be called when all OS threads on the thread pool have been suspended.
```

Note A thread pool cannot be suspended from an HPX thread running on the pool itself.

Parameters

- `callback`: [in] called when the thread pool has been suspended.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

Exceptions

- `hpx::exception`: if called from an HPX thread which is running on the pool itself.

thread_pools

See *Public API* for a list of names and headers that are part of the public *HPX API*.

thread_support

See *Public API* for a list of names and headers that are part of the public *HPX API*.

Defines

```
HPX_ASSERT_OWNS_LOCK(l)
HPX_ASSERT_DOESNT OWN_LOCK(l)
namespace hpx

namespace util
```

class atomic_count

Public Functions

```
HPX_NON_COPYABLE(atomic_count)
atomic_count(long value)
atomic_count &operator=(long value)
long operator++()
long operator--()
atomic_count &operator+=(long n)
atomic_count &operator-=(long n)
operator long() const
```

Private Members

```
    std::atomic<long> value_
namespace hpx

namespace util
```

Functions

```
void set_thread_name(char const*)
```

Defines

```
HPX_CORE_EXPORT_THREAD_SPECIFIC_PTR
namespace hpx
```

```
namespace util
```

```
template<typename T, typename Tag>
struct thread_specific_ptr
```

Public Types

```
typedef T element_type
```

Public Functions

```
T *get() const
T *operator->() const
T &operator*() const
void reset(T *new_value = nullptr)
```

Private Static Attributes

```
thread_local T *ptr_ = nullptr
namespace hpx

namespace util
```

```
template<typename Mutex>
class unlock_guard
```

Public Types

```
template<>
using mutex_type = Mutex
```

Public Functions

```
HPX_NON_COPYABLE(unlock_guard)
unlock_guard(Mutex &m)
~unlock_guard()
```

Private Members

```
Mutex &m_
```

threading

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

Functions

```
void swap(jthread &lhs, jthread &rhs)
class jthread
```

Public Types

```
using id = thread::id
using native_handle_type = thread::native_handle_type
```

Public Functions

```
jthread()
template<typename F, typename ...Ts, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<
```

```
jthread(F &&f, Ts&&... ts)
```

```
~jthread()
jthread(jthread const&)
jthread(jthread &&x)
jthread &operator=(jthread const&)
jthread &operator=(jthread&&)
void swap(jthread &t)
```

```

HPX_NODISCARD bool hpx::jthread::joinable() const
void join()
void detach()
HPX_NODISCARD id hpx::jthread::get_id() const
HPX_NODISCARD native_handle_type hpx::jthread::native_handle()
HPX_NODISCARD stop_source hpx::jthread::get_stop_source()
HPX_NODISCARD stop_token hpx::jthread::get_stop_token() const
bool request_stop()

```

Public Static Functions

```
static HPX_NODISCARD unsigned int hpx::jthread::hardware_concurrency()
```

Private Members

```

stop_source ssource_
hpx::thread thread_ = {}

```

Private Static Functions

```

template<typename F, typename ...Ts>
static void invoke(std::false_type, F &&f, stop_token&&, Ts&&... ts)
template<typename F, typename ...Ts>
static void invoke(std::true_type, F &&f, stop_token &&st, Ts&&... ts)
template<>
struct hash<::hpx::thread::id>

```

Public Functions

```

std::size_t operator() (::hpx::thread::id const &id) const
namespace hpx

```

TypeDefs

```

using thread_termination_handler_type = util::function_nonser<void (std::exception_ptr
const &e)>

```

Functions

```
void set_thread_termination_handler (thread_termination_handler_type f)
void swap (thread &x, thread &y)
bool operator== (thread::id const &x, thread::id const &y)
bool operator!= (thread::id const &x, thread::id const &y)
bool operator< (thread::id const &x, thread::id const &y)
bool operator> (thread::id const &x, thread::id const &y)
bool operator<= (thread::id const &x, thread::id const &y)
bool operator>= (thread::id const &x, thread::id const &y)
template<typename Char, typename Traits>
std::basic_ostream<Char, Traits> &operator<< (std::basic_ostream<Char, Traits> &out, thread::id
const &id)
class thread
```

Public Types

```
typedef threads::thread_id_type native_handle_type
```

Public Functions

```
thread ()
template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, th
thread (F &&f)
template<typename F, typename ...Ts>
thread (F &&f, Ts&&... vs)

template<typename F>
thread (threads::thread_pool_base *pool, F &&f)

template<typename F, typename ...Ts>
thread (threads::thread_pool_base *pool, F &&f, Ts&&... vs)

~thread ()
thread (thread&&)
thread &operator= (thread&&)
void swap (thread&)
bool joinable () const
void join ()
void detach ()
```

```

id get_id() const

native_handle_type native_handle() const

void interrupt (bool flag = true)

bool interruption_requested() const

hpx::future<void> get_future (error_code &ec = throws)

std::size_t get_thread_data() const

std::size_t set_thread_data (std::size_t)

```

Public Static Functions

```

static HPX_NODISCARD unsigned int hpx::thread::hardware_concurrency()

static void interrupt (id, bool flag = true)

```

Private Types

```
typedef lcos::local::spinlock mutex_type
```

Private Functions

```

void terminate (const char *function, const char *reason) const

bool joinable_locked() const

void detach_locked()

void start_thread (threads::thread_pool_base *pool, util::unique_function_nonser<void>
    > &&func)

```

Private Members

```

mutex_type mtx_

threads::thread_id_ref_type id_

```

Private Static Functions

```

static threads::thread_result_type thread_function_nullary (util::unique_function_nonser<void>
    > const &func

class id

```

Public Functions

```
id()
id(threads::thread_id_type const &i)
id(threads::thread_id_type &&i)
id(threads::thread_id_ref_type const &i)
id(threads::thread_id_ref_type &&i)
threads::thread_id_type const &native_handle() const
```

Private Members

```
threads::thread_id_type id_
```

Friends

```
friend hpx::thread

bool operator==(thread::id const &x, thread::id const &y)
bool operator!=(thread::id const &x, thread::id const &y)
bool operator<(thread::id const &x, thread::id const &y)
bool operator>(thread::id const &x, thread::id const &y)
bool operator<=(thread::id const &x, thread::id const &y)
bool operator>=(thread::id const &x, thread::id const &y)

template<typename Char, typename Traits>
std::basic_ostream<Char, Traits> &operator<<(std::basic_ostream<Char, Traits> &out,
                                                thread::id const &id)

namespace this_thread
```

Functions

```
thread::id get_id()

void yield()

void yield_to(thread::id)

threads::thread_priority get_priority()

std::ptrdiff_t get_stack_size()

void interruption_point()

bool interruption_enabled()

bool interruption_requested()
```

```
void interrupt ()  
void sleep_until (hpx::chrono::steady_time_point const &abs_time)  
void sleep_for (hpx::chrono::steady_duration const &rel_time)  
std::size_t get_thread_data ()  
std::size_t set_thread_data (std::size_t)  
class disable_interruption
```

Public Functions

```
disable_interruption ()  
~disable_interruption ()
```

Private Functions

```
disable_interruption (disable_interruption const&)  
disable_interruption &operator= (disable_interruption const&)
```

Private Members

```
bool interruption_was_enabled_
```

Friends

```
friend hpx::this_thread::restore_interruption  
class restore_interruption
```

Public Functions

```
restore_interruption (disable_interruption &d)  
~restore_interruption ()
```

Private Functions

```
restore_interruption (restore_interruption const&)  
restore_interruption &operator= (restore_interruption const&)
```

Private Members

```
    bool interruption_was_enabled_  
  
namespace std  
  
template<>  
struct hash<::hpx::thread::id>
```

Public Functions

```
std::size_t operator() (::hpx::thread::id const &id) const
```

threading_base

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

Functions

```
template<typename F>  
constexpr F &&annotated_function(F &&f, char const* = nullptr)
```

Returns a function annotated with the given annotation.

Annotating includes setting the thread description per thread id.

Parameters

- function:

```
template<typename F>  
constexpr F &&annotated_function(F &&f, std::string const&)
```

namespace util

Functions

```
template<typename F>  
constexpr decltype(auto) annotated_function(F &&f, char const *name = nullptr)
```

```
template<typename F>  
constexpr decltype(auto) annotated_function(F &&f, std::string const &name)
```

namespace hpx

namespace threads

namespace policies

```
class callback_notifier
```

Public Types

```
typedef util::function_nonser<void (std::size_t, std::size_t, char const*, char const*)>
    on_startstop_type

typedef util::function_nonser<bool (std::size_t,      std::exception_ptr      const&)>
    on_error_type
```

Public Functions

```
callback_notifier()

void on_start_thread(std::size_t local_thread_num, std::size_t global_thread_num,
                      char const *pool_name, char const *postfix) const

void on_stop_thread(std::size_t local_thread_num, std::size_t global_thread_num, char
                      const *pool_name, char const *postfix) const

bool on_error(std::size_t global_thread_num, std::exception_ptr const &e) const

void add_on_start_thread_callback(on_startstop_type const &callback)

void add_on_stop_thread_callback(on_startstop_type const &callback)

void set_on_error_callback(on_error_type const &callback)
```

Public Members

```
std::deque<on_startstop_type> on_start_thread_callbacks_
std::deque<on_startstop_type> on_stop_thread_callbacks_
on_error_type on_error_

namespace hpx
```

```
namespace threads
```

```
struct execution_agent : public agent_base
```

Public Functions

```
execution_agent (coroutines::detail::coroutine_impl *coroutine)

std::string description() const

execution_context const &context() const

void yield(char const *desc)

void yield_k(std::size_t k, char const *desc)

void suspend(char const *desc)

void resume(char const *desc)
```

```
void abort (char const *desc)
void sleep_for (hpx::chrono::steady_duration const &sleep_duration, char const *desc)
void sleep_until (hpx::chrono::steady_time_point const &sleep_time, char const
                  *desc)
```

Private Functions

```
hpx::threads::thread_restart_state do_yield (char const *desc,
                                                threads::thread_schedule_state state)
void do_resume (char const *desc, hpx::threads::thread_restart_state statex)
```

Private Members

```
coroutines::detail::coroutine_stackful_self self_
execution_context context_

struct execution_context : public context_base
```

Public Functions

```
hpx::execution_base::resource_base const &resource () const
```

Public Members

```
hpx::execution_base::resource_base resource_
namespace hpx

namespace util

namespace external_timer
```

Functions

```
std::shared_ptr<task_wrapper> new_task (thread_description const&, std::uint32_t,
                                             threads::thread_id_type)
std::shared_ptr<task_wrapper> update_task (std::shared_ptr<task_wrapper>,
                                              thread_description const&)
struct scoped_timer
```

Public Functions

```
scoped_timer(std::shared_ptr<task_wrapper>)

~scoped_timer()

void stop(void)

void yield(void)

namespace hpx
```

```
namespace threads
```

Functions

```
template<typename F>
thread_function_type make_thread_function(F &&f)

template<typename F>
thread_function_type make_thread_function_nullary(F &&f)

threads::thread_id_ref_type register_thread(threads::thread_init_data           &data,
                                             threads::thread_pool_base *pool, error_code
                                             &ec = throws)
```

Create a new *thread* using the given data.

Return This function will return the internal id of the newly created HPX-thread.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *data*: [in] The data to use for creating the thread.
- *pool*: [in] The thread pool to use for launching the work.
- *ec*: [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Exceptions

- *invalid_status*: if the runtime system has not been started yet.

```
threads::thread_id_ref_type register_thread(threads::thread_init_data &data, error_code
                                           &ec = throws)
```

Create a new *thread* using the given data on the same thread pool as the calling thread, or on the default thread pool if not on an HPX thread.

Return This function will return the internal id of the newly created HPX-thread.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *data*: [in] The data to use for creating the thread.
- *ec*: [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Exceptions

- *invalid_status*: if the runtime system has not been started yet.

```
thread_id_ref_type register_work(threads::thread_init_data &data, threads::thread_pool_base  
*pool, error_code &ec = throws)
```

Create a new work item using the given data.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *data*: [in] The data to use for creating the thread.
- *pool*: [in] The thread pool to use for launching the work.
- *ec*: [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Exceptions

- *invalid_status*: if the runtime system has not been started yet.

```
thread_id_ref_type register_work(threads::thread_init_data &data, error_code &ec = throws)
```

Create a new work item using the given data on the same thread pool as the calling thread, or on the default thread pool if not on an HPX thread.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *data*: [in] The data to use for creating the thread.
- *ec*: [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Exceptions

- *invalid_status*: if the runtime system has not been started yet.

namespace hpx

namespace threads

namespace policies

Functions

```
std::ostream &operator<<(std::ostream &os, scheduler_base const &scheduler)
```

struct scheduler_base

#include <scheduler_base.hpp> The *scheduler_base* defines the interface to be implemented by all scheduler policies

Public Types

```
typedef std::mutex pu_mutex_type  
using polling_function_ptr = detail::polling_status (*)()  
using polling_work_count_function_ptr = std::size_t (*)()
```

Public Functions

```

HPX_NON_COPYABLE (scheduler_base)

scheduler_base (std::size_t num_threads, char const *description = "",  

                  thread_queue_init_parameters thread_queue_init = {}, scheduler_mode mode = nothing_special)

virtual ~scheduler_base ()

threads::thread_pool_base *get_parent_pool () const

void set_parent_pool (threads::thread_pool_base *p)

std::size_t global_to_local_thread_index (std::size_t n)

std::size_t local_to_global_thread_index (std::size_t n)

char const *get_description () const

void idle_callback (std::size_t num_thread)

void do_some_work (std::size_t)

This function gets called by the thread-manager whenever new work has been added, allowing  

the scheduler to reactivate one or more of possibly idling OS threads

virtual void suspend (std::size_t num_thread)

virtual void resume (std::size_t num_thread)

std::size_t select_active_pu (std::unique_lock<pu_mutex_type> &l, std::size_t  

                           num_thread, bool allow_fallback = false)

std::atomic<hpx::state> &get_state (std::size_t num_thread)

std::atomic<hpx::state> const &get_state (std::size_t num_thread) const

void set_all_states (hpx::state s)

void set_all_states_at_least (hpx::state s)

bool has_reached_state (hpx::state s) const

bool is_state (hpx::state s) const

std::pair<hpx::state, hpx::state> get_minmax_state () const

scheduler_mode get_scheduler_mode () const

bool has_scheduler_mode (scheduler_mode mode) const

virtual void set_scheduler_mode (scheduler_mode mode)

void add_scheduler_mode (scheduler_mode mode)

void remove_scheduler_mode (scheduler_mode mode)

void add_remove_scheduler_mode (scheduler_mode to_add_mode, scheduler_mode  

                               to_remove_mode)

void update_scheduler_mode (scheduler_mode mode, bool set)

```

```
pu_mutex_type &get_pu_mutex (std::size_t num_thread)

std::size_t domain_from_local_thread_index (std::size_t n)

std::size_t num_domains (const std::size_t workers)

std::vector<std::size_t> domain_threads (std::size_t local_id, const
                                         std::vector<std::size_t> &ts,
                                         util::function_nonser<bool> std::size_t,
                                         std::size_t
                                         > pred

virtual std::int64_t get_queue_length (std::size_t num_thread = std::size_t(-1))
                                      const = 0

virtual std::int64_t get_thread_count (thread_schedule_state state =
                                         thread_schedule_state::unknown,
                                         thread_priority priority =
                                         thread_priority::default_,
                                         std::size_t num_thread = std::size_t(-1), bool reset
                                         = false) const = 0

virtual bool is_core_idle (std::size_t num_thread) const = 0

std::int64_t get_background_thread_count ()

void increment_background_thread_count ()

void decrement_background_thread_count ()

virtual bool enumerate_threads (util::function_nonser<bool> thread_id_type
                                > const &f, thread_schedule_state state =
                                         thread_schedule_state::unknown const = 0

virtual void abort_all_suspended_threads () = 0

virtual bool cleanup_terminated (bool delete_all) = 0

virtual bool cleanup_terminated (std::size_t num_thread, bool delete_all) = 0

virtual void create_thread (thread_init_data &data, thread_id_ref_type *id, error_code &ec) = 0

virtual bool get_next_thread (std::size_t num_thread, bool running,
                             threads::thread_id_ref_type &thrd, bool enable_stealing) = 0

virtual void schedule_thread (threads::thread_id_ref_type thrd,
                             threads::thread_schedule_hint schedulehint, bool allow_fallback
                             = false, thread_priority priority =
                             thread_priority::normal) = 0

virtual void schedule_thread_last (threads::thread_id_ref_type thrd,
                                   threads::thread_schedule_hint schedulehint,
                                   bool allow_fallback = false, thread_priority priority =
                                   thread_priority::normal) = 0

virtual void destroy_thread (threads::thread_data *thrd) = 0

virtual bool wait_or_add_new (std::size_t num_thread, bool running, std::int64_t
                             &idle_loop_count, bool enable_stealing, std::size_t
                             &added) = 0
```

```

virtual void on_start_thread(std::size_t num_thread) = 0
virtual void on_stop_thread(std::size_t num_thread) = 0
virtual void on_error(std::size_t num_thread, std::exception_ptr const &e) = 0
virtual void reset_thread_distribution()

std::ptrdiff_t get_stack_size(threads::thread_stacksize stacksize) const
void set_mpi_polling_functions(polling_function_ptr mpi_func,
polling_work_count_function_ptr mpi_work_count_func)
void clear_mpi_polling_function()
void set_cuda_polling_functions(polling_function_ptr cuda_func,
polling_work_count_function_ptr cuda_work_count_func)
void clear_cuda_polling_function()
detail::polling_status custom_polling_function() const
std::size_t get_polling_work_count() const

```

Public Static Functions

```

static detail::polling_status null_polling_function()
static std::size_t null_polling_work_count_function()

```

Protected Attributes

```

util::cache_line_data<std::atomic<scheduler_mode>> mode_
std::vector<pu_mutex_type> suspend_mtxs_
std::vector<std::condition_variable> suspendconds_
std::vector<pu_mutex_type> pu_mtxs_
std::vector<std::atomic<hpx::state>> states_
char const *description_
thread_queue_init_parameters thread_queue_init_
threads::thread_pool_base *parent_pool_
std::atomic<std::int64_t> background_thread_count_
std::atomic<polling_function_ptr> polling_function_mpi_
std::atomic<polling_function_ptr> polling_function_cuda_
std::atomic<polling_work_count_function_ptr> polling_work_count_function_mpi_
std::atomic<polling_work_count_function_ptr> polling_work_count_function_cuda_

```

namespace hpx

```
namespace threads
```

```
namespace policies
```

Enums

enum scheduler_mode

This enumeration describes the possible modes of a scheduler.

Values:

nothing_special = 0x000

As the name suggests, this option can be used to disable all other options.

do_background_work = 0x001

The scheduler will periodically call a provided callback function from a special HPX thread to enable performing background-work, for instance driving networking progress or garbage-collect AGAS.

reduce_thread_priority = 0x002

The kernel priority of the os-thread driving the scheduler will be reduced below normal.

delay_exit = 0x004

The scheduler will wait for some unspecified amount of time before exiting the scheduling loop while being terminated to make sure no other work is being scheduled during processing the shutdown request.

fast_idle_mode = 0x008

Some schedulers have the capability to act as ‘embedded’ schedulers. In this case it needs to periodically invoke a provided callback into the outer scheduler more frequently than normal. This option enables this behavior.

enable_elasticity = 0x010

This option allows for the scheduler to dynamically increase and reduce the number of processing units it runs on. Setting this value not succeed for schedulers that do not support this functionality.

enable_stealing = 0x020

This option allows schedulers that support work thread/stealing to enable/disable it

enable_stealing_numa = 0x040

This option allows schedulersthat support it to disallow stealing between numa domains

assign_work_round_robin = 0x080

This option tells schedulersthat support it to add tasks round robin to queues on each core

assign_work_thread_parent = 0x100

This option tells schedulers that support it to add tasks round to the same core/queue that the parent task is running on

steal_high_priority_first = 0x200

This option tells schedulers that support it to always (try to) steal high priority tasks from other queues before finishing their own lower priority tasks

steal_after_local = 0x400

This option tells schedulers that support it to steal tasks only when their local queues are empty

enable_idle_backoff = 0x0800

This option allows for certain schedulers to explicitly disable exponential idle-back off

default_mode = *do_background_work* | *reduce_thread_priority* | *delay_exit* | *enable_stealing* | *enable_stealing*
 This option represents the default mode.

all_flags = *do_background_work* | *reduce_thread_priority* | *delay_exit* | *fast_idle_mode* | *enable_elasticity* | *en*

This enables all available options.

```
namespace hpx
```

Enums

```
enum state
Values:
state_invalid = -1
state_initialized = 0
first_valid_runtime_state = state_initialized
state_pre_startup = 1
state_startup = 2
state_pre_main = 3
state_starting = 4
state_running = 5
state_suspended = 6
state_pre_sleep = 7
state_sleeping = 8
state_pre_shutdown = 9
state_shutdown = 10
state_stopping = 11
state_terminating = 12
state_stopped = 13
last_valid_runtime_state = state_stopped
```

```
namespace hpx
```

```
struct scoped_annotation
```

Public Functions

```
HPX_NON_COPYABLE(scoped_annotation)

constexpr scoped_annotation(char const*)

template<typename F>
constexpr scoped_annotation(F&&)

~scoped_annotation()
```

```
namespace util
```

Typedefs

```
using instead = hpx::scoped_annotation

namespace hpx

namespace threads
```

Functions

`thread_data *get_self_id_data()`

The function `get_self_id_data` returns the data of the HPX thread id associated with the current thread (or nullptr if the current thread is not a HPX thread).

`thread_data *get_thread_id_data(thread_id_ref_type const &tid)`

`thread_data *get_thread_id_data(thread_id_type const &tid)`

`thread_self &get_self()`

The function `get_self` returns a reference to the (OS thread specific) self reference to the current HPX thread.

`thread_self *get_self_ptr()`

The function `get_self_ptr` returns a pointer to the (OS thread specific) self reference to the current HPX thread.

`thread_self_impl_type *get_ctx_ptr()`

The function `get_ctx_ptr` returns a pointer to the internal data associated with each coroutine.

`thread_self *get_self_ptr_checked(error_code &ec = throws)`

The function `get_self_ptr_checked` returns a pointer to the (OS thread specific) self reference to the current HPX thread.

`thread_id_type get_self_id()`

The function `get_self_id` returns the HPX thread id of the current thread (or zero if the current thread is not a HPX thread).

`thread_id_type get_parent_id()`

The function `get_parent_id` returns the HPX thread id of the current thread's parent (or zero if the current thread is not a HPX thread).

Note This function will return a meaningful value only if the code was compiled with `HPX_HAVE_THREAD_PARENT_REFERENCE` being defined.

`std::size_t get_parent_phase()`

The function `get_parent_phase` returns the HPX phase of the current thread's parent (or zero if the current thread is not a HPX thread).

Note This function will return a meaningful value only if the code was compiled with `HPX_HAVE_THREAD_PARENT_REFERENCE` being defined.

`std::ptrdiff_t get_self_stacksize()`

The function `get_self_stacksize` returns the stack size of the current thread (or zero if the current thread is not a HPX thread).

`thread_stacksize get_self_stacksize_enum()`

The function `get_self_stacksize_enum` returns the stack size of the /.

`std::uint32_t get_parent_locality_id()`

The function `get_parent_locality_id` returns the id of the locality of the current thread's parent (or zero if the current thread is not a HPX thread).

Note This function will return a meaningful value only if the code was compiled with `HPX_HAVE_THREAD_PARENT_REFERENCE` being defined.

`std::uint64_t get_self_component_id()`

The function `get_self_component_id` returns the lva of the component the current thread is acting on

Note This function will return a meaningful value only if the code was compiled with `HPX_HAVE_THREAD_TARGET_ADDRESS` being defined.

`class thread_data : public thread_data_reference_counting`

`#include <thread_data.hpp>` A *thread* is the representation of a ParalleX thread. It's a first class object in ParalleX. In our implementation this is a user level thread running on top of one of the OS threads spawned by the *thread-manager*.

A *thread* encapsulates:

- A thread status word (see the functions `thread::get_state` and `thread::set_state`)
- A function to execute (the `thread` function)
- A frame (in this implementation this is a block of memory used as the threads stack)
- A block of registers (not implemented yet)

Generally, *threads* are not created or executed directly. All functionality related to the management of *threads* is implemented by the *thread-manager*.

Subclassed by `hpx::threads::thread_data_stackful`, `hpx::threads::thread_data_stackless`

Public Types

`using spinlock_pool = util::spinlock_pool<thread_data>`

Public Functions

`thread_data(thread_data const&)`

`thread_data(thread_data&&)`

`thread_data &operator=(thread_data const&)`

`thread_data &operator=(thread_data&&)`

`thread_state get_state(std::memory_order order = std::memory_order_acquire) const`

The `get_state` function queries the state of this thread instance.

Return This function returns the current state of this thread. It will return one of the values as defined by the `thread_state` enumeration.

Note This function will be seldom used directly. Most of the time the state of a thread will be retrieved by using the function `threadmanager::get_state`.

```
thread_state set_state(thread_schedule_state state, thread_restart_state state_ex =
                     thread_restart_state::unknown, std::memory_order load_order =
                     std::memory_order_acquire, std::memory_order exchange_order =
                     std::memory_order_seq_cst)
```

The set_state function changes the state of this thread instance.

Note This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager. Moreover, changing the thread state using this function does not change its scheduling status. It only sets the thread's status word. To change the thread's scheduling status *threadmanager::set_state* should be used.

Parameters

- newstate: [in] The new state to be set for the thread.

```
bool set_state_tagged(thread_schedule_state newstate, thread_state &prev_state,
                      thread_state &new_tagged_state, std::memory_order exchange_order =
                      std::memory_order_seq_cst)
```

```
bool restore_state(thread_state new_state, thread_state old_state, std::memory_order load_order =
                     std::memory_order_relaxed, std::memory_order load_exchange =
                     std::memory_order_seq_cst)
```

The restore_state function changes the state of this thread instance depending on its current state. It will change the state atomically only if the current state is still the same as passed as the second parameter. Otherwise it won't touch the thread state of this instance.

Note This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager. Moreover, changing the thread state using this function does not change its scheduling status. It only sets the thread's status word. To change the thread's scheduling status *threadmanager::set_state* should be used.

Return This function returns *true* if the state has been changed successfully

Parameters

- newstate: [in] The new state to be set for the thread.
- oldstate: [in] The old state of the thread which still has to be the current state.

```
bool restore_state(thread_schedule_state new_state, thread_restart_state state_ex,
                    thread_state old_state, std::memory_order load_exchange =
                    std::memory_order_seq_cst)
```

constexpr std::uint64_t get_component_id() const

Return the id of the component this thread is running in.

util::thread_description get_description() const

util::thread_description set_description(util::thread_description)

util::thread_description get_lco_description() const

util::thread_description set_lco_description(util::thread_description)

constexpr std::uint32_t get_parent_locality_id() const

Return the locality of the parent thread.

constexpr thread_id_type get_parent_thread_id() const

Return the thread id of the parent thread.

constexpr std::size_t get_parent_thread_phase() const

Return the phase of the parent thread.

```

constexpr util::backtrace const *get_backtrace() const
util::backtrace const *set_backtrace(util::backtrace const*)

constexpr thread_priority get_priority() const
void set_priority(thread_priority priority)

bool interruption_requested() const
bool interruption_enabled() const
bool set_interruption_enabled(bool enable)
void interrupt(bool flag = true)
bool interruption_point(bool throw_on_interrupt = true)

bool add_thread_exit_callback(util::function_nonsr<void>
    > const &f)

void run_thread_exit_callbacks()

void free_thread_exit_callbacks()

bool is_stackless() const
void destroy_thread()

get_scheduler_base() const
std::size_t get_last_worker_thread_num() const
void set_last_worker_thread_num(std::size_t last_worker_thread_num)

std::ptrdiff_t get_stack_size() const
thread_stacksize get_stack_size_enum() const

template<typename ThreadQueue>
ThreadQueue &get_queue()

coroutine_type::result_type operator() (hpx::execution_base::this_thread::detail::agent_storage
    *agent_storage)
Execute the thread function.

```

Return This function returns the thread state the thread should be scheduled from this point on.
The thread manager will use the returned value to set the thread's scheduling status.

```

virtual thread_id_type get_thread_id() const
virtual std::size_t get_thread_phase() const
virtual std::size_t get_thread_data() const = 0
virtual std::size_t set_thread_data(std::size_t data) = 0
virtual void init() = 0
virtual void rebind(thread_init_data &init_data) = 0

```

```
thread_data(thread_init_data &init_data, void *queue, std::ptrdiff_t stacksize, bool  
is_stackless = false, thread_id_addr addref = thread_id_addr::yes)  
virtual ~thread_data()  
virtual void destroy() = 0
```

Protected Functions

```
thread_restart_state set_state_ex(thread_restart_state new_state)  
The set_state function changes the extended state of this thread instance.
```

Note This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager.

Parameters

- newstate: [in] The new extended state to be set for the thread.

```
void rebinding_base(thread_init_data &init_data)
```

Private Members

```
std::atomic<thread_state> current_state_  
thread_priority priority_  
bool requested_interrupt_  
bool enabled_interrupt_  
bool ran_exit_funcs_  
const bool is_stackless_  
std::forward_list<util::function_nonser<void ()>> exit_funcs_  
policies::scheduler_base *scheduler_base_  
std::size_t last_worker_thread_num_  
std::ptrdiff_t stacksize_  
thread_stacksize stacksize_enum_  
void *queue_  
namespace hpx
```

```
namespace threads
```

```
class thread_data_stackful : public hpx::threads::thread_data  
#include <thread_data_stackful.hpp> A thread is the representation of a ParalleX thread. It's a first class object in ParalleX. In our implementation this is a user level thread running on top of one of the OS threads spawned by the thread-manager.
```

A *thread* encapsulates:

- A thread status word (see the functions *thread::get_state* and *thread::set_state*)
- A function to execute (the thread function)
- A frame (in this implementation this is a block of memory used as the threads stack)

- A block of registers (not implemented yet)
- Generally, *threads* are not created or executed directly. All functionality related to the management of *threads* is implemented by the thread-manager.

Public Functions

```
coroutine_type::result_type call (hpx::execution_base::this_thread::detail::agent_storage
                                *agent_storage)

std::size_t get_thread_data () const

std::size_t set_thread_data (std::size_t data)

void init ()

void rebind (thread_init_data &init_data)

thread_data_stackful (thread_init_data &init_data, void *queue, std::ptrdiff_t stacksize,
                           thread_id_addr addref addref)

~thread_data_stackful ()

void destroy ()
```

Public Static Functions

```
thread_data *create (thread_init_data &init_data, void *queue, std::ptrdiff_t stacksize,
                           thread_id_addr addref addref = thread_id_addr::yes)
```

Private Functions

```
thread_data *this_ ()
```

Private Members

```
coroutine_type coroutine_
execution_agent agent_
```

Private Static Attributes

```
util::internal_allocator<thread_data_stackful> thread_alloc_
namespace hpx
```

```
namespace threads
```

```
class thread_data_stackless : public hpx::threads::thread_data
#include <thread_data_stackless.hpp> A thread is the representation of a ParalleX thread. It's a first
class object in ParalleX. In our implementation this is a user level thread running on top of one of the
OS threads spawned by the thread-manager.
```

A *thread* encapsulates:

- A thread status word (see the functions `thread::get_state` and `thread::set_state`)
- A function to execute (the thread function)
- A frame (in this implementation this is a block of memory used as the threads stack)
- A block of registers (not implemented yet)

Generally, `threads` are not created or executed directly. All functionality related to the management of `threads` is implemented by the thread-manager.

Public Functions

```
stackless_coroutine_type::result_type call ()  
  
std::size_t get_thread_data () const  
  
std::size_t set_thread_data (std::size_t data)  
  
void init ()  
  
void rebind (thread_init_data &init_data)  
  
thread_data_stackless (thread_init_data &init_data, void *queue, std::ptrdiff_t stacksize, thread_id_addr addref addref)  
  
~thread_data_stackless ()  
  
void destroy ()
```

Public Static Functions

```
thread_data *create (thread_init_data &data, void *queue, std::ptrdiff_t stacksize, thread_id_addr addref addref = thread_id_addr::yes)
```

Private Functions

```
thread_data *this_ ()
```

Private Members

```
stackless_coroutine_type coroutine_
```

Private Static Attributes

```
util::internal_allocator<thread_data_stackless> thread_alloc_  
namespace hpx  
  
namespace threads
```

Functions

```
util::thread_description get_thread_description(thread_id_type const &id, error_code  
&ec = throws)
```

The function `get_thread_description` is part of the thread related API allows to query the description of one of the threads known to the thread-manager.

Return This function returns the description of the thread referenced by the `id` parameter. If the thread is not known to the thread-manager the return value will be the string “<unknown>”.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn’t throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread being queried.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
util::thread_description set_thread_description(thread_id_type const &id,  
util::thread_description const &desc  
= util::thread_description(), error_code  
&ec = throws)
```

```
util::thread_description get_thread_lco_description(thread_id_type const &id, error_code &ec = throws)
```

```
util::thread_description set_thread_lco_description(thread_id_type const &id,  
util::thread_description const &desc = util::thread_description(),  
error_code &ec = throws)
```

```
namespace util
```

Functions

```
std::ostream &operator<<(std::ostream&, thread_description const&)
```

```
std::string as_string(thread_description const &desc)
```

```
struct thread_description
```

Public Types

```
enum data_type
```

Values:

```
    data_type_description = 0
```

```
    data_type_address = 1
```

Public Functions

```
thread_description()

constexpr thread_description(char const*)

template<typename F, typename = typename std::enable_if<!std::is_same<F, thread_description>::value && !traits>
constexpr thread_description(F const&, char const* = nullptr)

template<typename Action, typename = typename std::enable_if<traits::is_action<Action>::value>::type>
constexpr thread_description(Action, char const* = nullptr)

constexpr data_type kind() const

constexpr char const *get_description() const

constexpr std::size_t get_address() const

constexpr operator bool() const

constexpr bool valid() const
```

Private Functions

```
void init_from_alternative_name(char const *altname)

namespace hpx

namespace this_thread
```

Functions

```
threads::thread_restart_state suspend(threads::thread_schedule_state
                                      state,                      threads::thread_id_type      id,
                                      util::thread_description const &description = util::thread_description("this_thread::suspend"),
                                      util::error_code &ec = throws)
```

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to the thread state passed as the parameter.

Note Must be called from within a HPX-thread.

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
threads::thread_restart_state suspend(threads::thread_schedule_state      state      =
                                         threads::thread_schedule_state::pending,
                                         util::thread_description const &description = 
                                         util::thread_description("this_thread::suspend"),
                                         error_code &ec = throws)
```

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to the thread state passed as the parameter.

Note Must be called from within a HPX-thread.

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
threads::thread_restart_state suspend(hpx::chrono::steady_time_point      const
                                         &abs_time,           threads::thread_id_type id,
                                         util::thread_description const &description = 
                                         util::thread_description("this_thread::suspend"),
                                         error_code &ec = throws)
```

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads at the given time.

Note Must be called from within a HPX-thread.

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
threads::thread_restart_state suspend(hpx::chrono::steady_time_point const &abs_time,
                                         util::thread_description const &description = 
                                         util::thread_description("this_thread::suspend"),
                                         error_code &ec = throws)
```

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads at the given time.

Note Must be called from within a HPX-thread.

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
threads::thread_restart_state suspend(hpx::chrono::steady_duration const &rel_time,
                                         util::thread_description const &description = 
                                         util::thread_description("this_thread::suspend"),
                                         error_code &ec = throws)
```

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this 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_restart_state suspend(hpx::chrono::steady_duration      const      &rel_time,
                                         threads::thread_id_type        const      &id,
                                         util::thread_description      const      &description = =
                                         util::thread_description("this_thread::suspend"),         er-
                                         ror_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_restart_state suspend(std::uint64_t ms, util::thread_description const &descrip-
                                         tion = util::thread_description("this_thread::suspend"), er-
                                         ror_code &ec = throws)
```

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads after the given time (specified in milliseconds).

Note Must be called from within a HPX-thread.

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
threads::thread_pool_base *get_pool(error_code &ec = throws)
```

Returns a pointer to the pool that was used to run the current thread

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of

hpx::yield_aborted if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::invalid_status*.

namespace threads

Functions

```
thread_state set_thread_state(thread_id_type const &id, thread_schedule_state state
= thread_schedule_state::pending, thread_restart_state stateex =
thread_restart_state::signaled, thread_priority priority =
thread_priority::normal, bool retry_on_active = true,


```

Set the thread state of the *thread* referenced by the thread_id *id*.

Note If the thread referenced by the parameter *id* is in *thread_state::active* state this function schedules a new thread which will set the state of the thread as soon as its not active anymore. The function returns *thread_state::active* in this case.

Return This function returns the previous state of the thread referenced by the *id* parameter. It will return one of the values as defined by the *thread_state* enumeration. If the thread is not known to the thread-manager the return value will be *thread_state::unknown*.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *id*: [in] The thread id of the thread the state should be modified for.
- *state*: [in] The new state to be set for the thread referenced by the *id* parameter.
- *stateex*: [in] The new extended state to be set for the thread referenced by the *id* parameter.
- *priority*:
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
thread_id_ref_type set_thread_state(thread_id_type const &id,
hpx::chrono::steady_time_point const &abs_time,
std::atomic<bool> *started, thread_schedule_state state
= thread_schedule_state::pending, thread_restart_state
stateex = thread_restart_state::timeout, thread_priority
priority = thread_priority::normal, bool retry_on_active
= true, error_code &ec = throws)
```

Set the thread state of the *thread* referenced by the thread_id *id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (at the given time)

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_ref_type set_thread_state(thread_id_type const &id,  
                                hpx::chrono::steady_time_point const  
                                &abs_time,      thread_schedule_state state =  
                                thread_schedule_state::pending,   thread_restart_state  
                                stateex = thread_restart_state::timeout, thread_priority  
                                priority = thread_priority::normal, bool retry_on_active  
                                = true, error_code& = throws)
```

```
thread_id_ref_type set_thread_state(thread_id_type const &id,  
                                hpx::chrono::steady_duration const  
                                &rel_time,      thread_schedule_state state =  
                                thread_schedule_state::pending,   thread_restart_state  
                                stateex = thread_restart_state::timeout, thread_priority  
                                priority = thread_priority::normal, bool retry_on_active  
                                = true, error_code &ec = throws)
```

Set the thread state of the *thread* referenced by the *thread_id id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (after the given duration)

Return

Note As long as *ec* is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread the state should be modified for.
- `rel_time`: [in] Time duration after which the new thread should be run
- `state`: [in] The new state to be set for the thread referenced by the *id* parameter.
- `stateex`: [in] The new extended state to be set for the thread referenced by the *id* parameter.
- `priority`:
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
thread_state get_thread_state(thread_id_type const &id, error_code &ec = throws)
```

The function `get_thread_backtrace` is part of the thread related API allows to query the currently stored thread back trace (which is captured during thread suspension).

Return This function returns the currently captured stack back trace of the thread referenced by the *id* parameter. If the thread is not known to the thread-manager the return value will be the zero.

Note As long as *ec* is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`. The function `get_thread_state` is part of the thread related API. It queries the state of one of the threads known to the thread-manager.

Return This function returns the thread state of the thread referenced by the *id* parameter. If the thread is not known to the thread-manager the return value will be *terminated*.

Note As long as *ec* is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread being queried.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

Parameters

- `id`: [in] The thread id of the thread the state should be modified for.

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`std::size_t get_thread_phase(thread_id_type const &id, error_code &ec = throws)`

The function `get_thread_phase` is part of the thread related API. It queries the phase of one of the threads known to the thread-manager.

Return This function returns the thread phase of the thread referenced by the `id` parameter. If the thread is not known to the thread-manager the return value will be `~0`.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread the phase should be modified for.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`bool get_thread_interruption_enabled(thread_id_type const &id, error_code &ec = throws)`

Returns whether the given thread can be interrupted at this point.

Return This function returns `true` if the given thread can be interrupted at this point in time. It will return `false` otherwise.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread which should be queried.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`bool set_thread_interruption_enabled(thread_id_type const &id, bool enable, error_code &ec = throws)`

Set whether the given thread can be interrupted at this point.

Return This function returns the previous value of whether the given thread could have been interrupted.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread which should receive the new value.
- `enable`: [in] This value will determine the new interruption enabled status for the given thread.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`bool get_thread_interruption_requested(thread_id_type const &id, error_code &ec = throws)`

Returns whether the given thread has been flagged for interruption.

Return This function returns `true` if the given thread was flagged for interruption. It will return `false` otherwise.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread which should be queried.

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
void interrupt_thread(thread_id_type const &id, bool flag, error_code &ec = throws)  
Flag the given thread for interruption.
```

Note As long as *ec* is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread which should be interrupted.
- `flag`: [in] The flag encodes whether the thread should be interrupted (if it is *true*), or ‘uninterrupted’ (if it is *false*).
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
void interrupt_thread(thread_id_type const &id, error_code &ec = throws)
```

```
void interruption_point(thread_id_type const &id, error_code &ec = throws)
```

Interrupt the current thread at this point if it was canceled. This will throw a `thread_interrupted` exception, which will cancel the thread.

Note As long as *ec* is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread which should be interrupted.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
threads::thread_priority get_thread_priority(thread_id_type const &id, error_code &ec =  
                                  throws)
```

Return priority of the given thread

Note As long as *ec* is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread whose priority is queried.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
std::ptrdiff_t get_stack_size(thread_id_type const &id, error_code &ec = throws)
```

Return stack size of the given thread

Note As long as *ec* is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `id`: [in] The thread id of the thread whose priority is queried.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
threads::thread_pool_base *get_pool(thread_id_type const &id, error_code &ec = throws)
```

Returns a pointer to the pool that was used to run the current thread

Exceptions

- If: `&ec != &throws`, never throws, but will set `ec` to an appropriate value when an error occurs. Otherwise, this function will throw an `hpx::exception` with an error code of `hpx::yield_aborted` if it is signaled with `wait_aborted`. If called outside of a HPX-thread, this function will throw an `hpx::exception` with an error code of `hpx::null_thread_id`. If this function is called while the thread-manager is not running, it will throw an `hpx::exception` with an error code of `hpx::invalid_status`.

```
namespace hpx
```

```
namespace threads
```

```
class thread_init_data
```

Public Functions

```
thread_init_data()  

thread_init_data &operator=(thread_init_data &&rhs)  

thread_init_data(thread_init_data &&rhs)  

template<typename F>  

thread_init_data(F &&f, util::thread_description const &desc, thread_priority  

priority_ = thread_priority::normal, thread_schedule_hint  

os_thread = thread_schedule_hint(), thread_stacksize stacksize_  

= thread_stacksize::default_, thread_schedule_state initial_state_  

= thread_schedule_state::pending, bool run_now_ = false, policies::scheduler_base *scheduler_base_ = nullptr)
```

Public Members

```
threads::thread_function_type func  

thread_priority priority  

thread_schedule_hint schedulehint  

thread_stacksize stacksize  

thread_schedule_state initial_state  

bool run_now  

policies::scheduler_base *scheduler_base
```

```
namespace hpx
```

Functions

`std::size_t get_worker_thread_num()`

Return the number of the current OS-thread running in the runtime instance the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the overall number of OS-threads executed (as returned by `get_os_thread_count()`).

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

`std::size_t get_worker_thread_num(error_code &ec)`

Return the number of the current OS-thread running in the runtime instance the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the overall number of OS-threads executed (as returned by `get_os_thread_count()`). It will return -1 if the current thread is not a known thread or if the runtime is not in running state.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters

- `ec`: [in,out] this represents the error status on exit.

`std::size_t get_local_worker_thread_num()`

Return the number of the current OS-thread running in the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread on the current thread pool which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the number of OS-threads executed on the current thread pool. It will return -1 if the current thread is not a known thread or if the runtime is not in running state.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

`std::size_t get_local_worker_thread_num(error_code &ec)`

Return the number of the current OS-thread running in the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread on the current thread pool which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the number of OS-threads executed on the current thread pool. It will return -1 if the current thread is not a known thread or if the runtime is not in running state.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters

- `ec`: [in,out] this represents the error status on exit.

`std::size_t get_thread_pool_num()`

Return the number of the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the thread pool which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the number of thread pools started by the runtime. It will return -1 if the current thread pool is not a known thread pool or if the runtime is not in running state.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

`std::size_t get_thread_pool_num(error_code &ec)`

Return the number of the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the thread pool which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the number of thread pools started by the runtime. It will return -1 if the current thread pool is not a known thread pool or if the runtime is not in running state.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters

- `ec`: [in,out] this represents the error status on exit.

`namespace hpx`

`namespace threads`

Functions

`std::ostream &operator<< (std::ostream &os, thread_pool_base const &thread_pool)`

`struct executor_statistics`

`#include <thread_pool_base.hpp>` Data structure which stores statistics collected by an executor instance.

Public Functions

`executor_statistics()`

Public Members

```
std::uint64_t tasks_scheduled_
std::uint64_t tasks_completed_
std::uint64_t queue_length_
```

class `thread_pool_base`

#include <thread_pool_base.hpp> The base class used to manage a pool of OS threads.

Public Functions

virtual void suspend_processing_unit_direct (`std::size_t virt_core, error_code &ec = throws`) = 0

Suspends the given processing unit. Blocks until the processing unit has been suspended.

Parameters

- `virt_core`: [in] The processing unit on the the pool to be suspended. The processing units are indexed starting from 0.

virtual void resume_processing_unit_direct (`std::size_t virt_core, error_code &ec = throws`) = 0

Resumes the given processing unit. Blocks until the processing unit has been resumed.

Parameters

- `virt_core`: [in] The processing unit on the the pool to be resumed. The processing units are indexed starting from 0.

virtual void resume_direct (`error_code &ec = throws`) = 0

Resumes the thread pool. Blocks until all OS threads on the thread pool have been resumed.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

virtual void suspend_direct (`error_code &ec = throws`) = 0

Suspends the thread pool. Blocks until all OS threads on the thread pool have been suspended.

Note A thread pool cannot be suspended from an HPX thread running on the pool itself.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

Exceptions

- `hpx::exception`: if called from an HPX thread which is running on the pool itself.

struct `thread_pool_init_parameters`

Public Functions

```
thread_pool_init_parameters(std::string const &name, std::size_t index,
policies::scheduler_mode mode, std::size_t num_threads, std::size_t thread_offset,
hpx::threads::policies::callback_notifier &notifier, hpx::threads::policies::detail::affinity_data const &affinity_data,
hpx::threads::detail::network_background_callback_type const &network_background_callback =
hpx::threads::detail::network_background_callback_type(),
std::size_t max_background_threads = std::size_t(-1), std::size_t max_idle_loop_count =
HPX_IDLE_LOOP_COUNT_MAX, std::size_t max_busy_loop_count =
HPX_BUSY_LOOP_COUNT_MAX, std::size_t shutdown_check_count = 10)
```

Public Members

```
std::string const &name_
std::size_t index_
policies::scheduler_mode mode_
std::size_t num_threads_
std::size_t thread_offset_
hpx::threads::policies::callback_notifier &notifier_
hpx::threads::policies::detail::affinity_data const &affinity_data_
hpx::threads::detail::network_background_callback_type const &network_background_callback_
std::size_t max_background_threads_
std::size_t max_idle_loop_count_
std::size_t max_busy_loop_count_
std::size_t shutdown_check_count_

namespace hpx

namespace threads

namespace policies

struct thread_queue_init_parameters
```

Public Functions

```
thread_queue_init_parameters (std::int64_t      max_thread_count      =
                             std::int64_t(HPX_THREAD_QUEUE_MAX_THREAD_COUNT),
                             std::int64_t      min_tasks_to_stal_pending      =
                             std::int64_t(HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_PENDING),
                             std::int64_t      min_tasks_to_stal_staged      =
                             std::int64_t(HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_STAGED),
                             std::int64_t      min_add_new_count      =
                             std::int64_t(HPX_THREAD_QUEUE_MIN_ADD_NEW_COUNT),
                             std::int64_t      max_add_new_count      =
                             std::int64_t(HPX_THREAD_QUEUE_MAX_ADD_NEW_COUNT),
                             std::int64_t      min_delete_count      =
                             std::int64_t(HPX_THREAD_QUEUE_MIN_DELETE_COUNT),
                             std::int64_t      max_delete_count      =
                             std::int64_t(HPX_THREAD_QUEUE_MAX_DELETE_COUNT),
                             std::int64_t      max_terminated_threads      =
                             std::int64_t(HPX_THREAD_QUEUE_MAX_TERMINATED_THREADS),
                             std::int64_t      init_threads_count      =
                             std::int64_t(HPX_THREAD_QUEUE_INIT_THREADS_COUNT),
                             double      max_idle_backoff_time      =      dou-
                                ble(HPX_IDLE_BACKOFF_TIME_MAX),
                             std::ptrdiff_t      small_stacksize      =
                             HPX_SMALL_STACK_SIZE,
                             std::ptrdiff_t      medium_stacksize      =
                             HPX_MEDIUM_STACK_SIZE,      std::ptrdiff_t
                             large_stacksize = HPX_LARGE_STACK_SIZE,
                             std::ptrdiff_t      huge_stacksize      =
                             HPX_HUGE_STACK_SIZE)
```

Public Members

```
std::int64_t max_thread_count_
std::int64_t min_tasks_to_stal_pending_
std::int64_t min_tasks_to_stal_staged_
std::int64_t min_add_new_count_
std::int64_t max_add_new_count_
std::int64_t min_delete_count_
std::int64_t max_delete_count_
std::int64_t max_terminated_threads_
std::int64_t init_threads_count_
double max_idle_backoff_time_
const std::ptrdiff_t small_stacksize_
const std::ptrdiff_t medium_stacksize_
const std::ptrdiff_t large_stacksize_
const std::ptrdiff_t huge_stacksize_
```

```
const std::ptrdiff_t nostack_stacksize_
```

threadmanager

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace threads
```

```
class threadmanager
```

```
#include <threadmanager.hpp> The thread-manager class is the central instance of management for all (non-depleted) threads
```

Public Types

```
typedef threads::policies::callback_notifier notification_policy_type  

typedef std::unique_ptr<thread_pool_base> pool_type  

typedef threads::policies::scheduler_base scheduler_type  

typedef std::vector<pool_type> pool_vector
```

Public Functions

```
threadmanager (hpx::util::runtime_configuration &rtefg_, notification_policy_type  

&notifier, detail::network_background_callback_type net-  

work_background_callback = detail::network_background_callback())  

~threadmanager ()  

void init ()  

void create_pools ()  

void print_pools (std::ostream&)  

    FIXME move to private and add hpx:printpools cmd line option.  

thread_pool_base &default_pool () const  

scheduler_type &default_scheduler () const  

thread_pool_base &get_pool (std::string const &pool_name) const  

thread_pool_base &get_pool (pool_id_type const &pool_id) const  

thread_pool_base &get_pool (std::size_t thread_index) const  

bool pool_exists (std::string const &pool_name) const  

bool pool_exists (std::size_t pool_index) const
```

thread_id_ref_type register_work (thread_init_data &data, error_code &ec = throws)

The function *register_work* adds a new work item to the thread manager. It doesn't immediately create a new *thread*, it just adds the task parameters (function, initial state and description) to the internal management data structures. The thread itself will be created when the number of existing threads drops below the number of threads specified by the constructors *max_count* parameter.

Parameters

- *func*: [in] The function or function object to execute as the thread's function. This must have a signature as defined by *thread_function_type*.
- *description*: [in] The value of this parameter allows to specify a description of the thread to create. This information is used for logging purposes mainly, but might be useful for debugging as well. This parameter is optional and defaults to an empty string.

void register_thread (thread_init_data &data, thread_id_ref_type &id, error_code &ec = throws)

The function *register_thread* adds a new work item to the thread manager. It creates a new *thread*, adds it to the internal management data structures, and schedules the new thread, if appropriate.

Parameters

- *func*: [in] The function or function object to execute as the thread's function. This must have a signature as defined by *thread_function_type*.
- *id*: [out] This parameter will hold the id of the created thread. This id is guaranteed to be validly initialized before the thread function is executed.
- *description*: [in] The value of this parameter allows to specify a description of the thread to create. This information is used for logging purposes mainly, but might be useful for debugging as well. This parameter is optional and defaults to an empty string.

bool run ()

Run the thread manager's work queue. This function instantiates the specified number of OS threads in each pool. All OS threads are started to execute the function *tfunc*.

Return The function returns *true* if the thread manager has been started successfully, otherwise it returns *false*.

void stop (bool blocking = true)

Forcefully stop the thread-manager.

Parameters

- *blocking*:

bool is_busy ()

bool is_idle ()

void wait ()

void suspend ()

void resume ()

state status () const

Return whether the thread manager is still running. This returns the "minimal state", i.e. the state of the least advanced thread pool.

```
std::int64_t get_thread_count (thread_schedule_state state = thread_schedule_state::unknown, thread_priority priority = thread_priority::default_, std::size_t num_thread = std::size_t(-1), bool reset = false)
    return the number of HPX-threads with the given state
```

Note This function lock the internal OS lock in the thread manager

```
std::int64_t get_idle_core_count ()

mask_type get_idle_core_mask ()

std::int64_t get_background_thread_count ()

bool enumerate_threads (util::function_nonser<bool> thread_id_type
    > const &f, thread_schedule_state state = thread_schedule_state::unknown) const

void abort_all_suspended_threads ()

bool cleanup_terminated (bool delete_all)

std::size_t get_os_thread_count () const
    Return the number of OS threads running in this thread-manager.

    This function will return correct results only if the thread-manager is running.

std::thread &get_os_thread_handle (std::size_t num_thread) const

void report_error (std::size_t num_thread, std::exception_ptr const &e)
    API functions forwarding to notification policy.

    This notifies the thread manager that the passed exception has been raised. The exception will be
    routed through the notifier and the scheduler (which will result in it being passed to the runtime
    object, which in turn will report it to the console, etc.).

mask_type get_used_processing_units () const
    Returns the mask identifying all processing units used by this thread manager.

hwloc_bitmap_ptr get_pool numa_bitmap (const std::string &pool_name) const

void set_scheduler_mode (threads::policies::scheduler_mode mode)

void add_scheduler_mode (threads::policies::scheduler_mode mode)

void add_remove_scheduler_mode (threads::policies::scheduler_mode
    to_add_mode, threads::policies::scheduler_mode
    to_remove_mode)

void remove_scheduler_mode (threads::policies::scheduler_mode mode)

void reset_thread_distribution ()

void init_tss (std::size_t global_thread_num)

void deinit_tss ()

std::int64_t get_queue_length (bool reset)

std::int64_t get_cumulative_duration (bool reset)
```

```
std::int64_t get_thread_count_unknown(bool reset)
std::int64_t get_thread_count_active(bool reset)
std::int64_t get_thread_count_pending(bool reset)
std::int64_t get_thread_count_suspended(bool reset)
std::int64_t get_thread_count_terminated(bool reset)
std::int64_t get_thread_count_staged(bool reset)
```

Private Types

```
typedef std::mutex mutex_type
```

Private Members

```
mutex_type mtx_
hpx::util::runtime_configuration &rtcfg_
std::vector<pool_id_type> threads_lookup_
pool_vector pools_
notification_policy_type &notifier_
detail::network_background_callback_type network_background_callback_
```

timed_execution

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

Variables

```
hpx::parallel::execution::post_at_t post_at
hpx::parallel::execution::post_after_t post_after
hpx::parallel::execution::async_execute_at_t async_execute_at
hpx::parallel::execution::async_execute_after_t async_execute_after
hpx::parallel::execution::sync_execute_at_t sync_execute_at
hpx::parallel::execution::sync_execute_after_t sync_execute_after
```

```
struct async_execute_after_t : public hpx::functional::detail::tag_fallback<async_execute_after_t>
#include <timed_execution_fwd.hpp> Customization point of asynchronous execution agent creation supporting timed execution.
```

This asynchronously creates a single function invocation `f()` using the associated executor at the given point in time.

Return `f(ts...)`'s result through a future

Note This calls `exec.async_execute_after(rel_time, f, ts...)`, if available, otherwise it emulates timed scheduling by delaying calling `execution::async_execute()` on the underlying non-time-scheduled execution agent.

Parameters

- `exec`: [in] The executor object to use for scheduling of the function *f*.
- `rel_time`: [in] The duration of time after which the given function should be scheduled to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts... : [in]` Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
decltype(auto) friend tag_fallback_invoke(async_execute_after_t, Executor
                                         &&exec, hpx::chrono::steady_duration
                                         const &rel_time, F &&f, Ts&&... ts)
```

```
struct async_execute_at_t : public hpx::functional::detail::tag_fallback<async_execute_at_t>
#include <timed_execution_fwd.hpp> Customization point of asynchronous execution agent creation supporting timed execution.
```

This asynchronously creates a single function invocation `f()` using the associated executor at the given point in time.

Return `f(ts...)`'s result through a future

Note This calls `exec.async_execute_at(abs_time, f, ts...)`, if available, otherwise it emulates timed scheduling by delaying calling `execution::async_execute()` on the underlying non-time-scheduled execution agent.

Parameters

- `exec`: [in] The executor object to use for scheduling of the function *f*.
- `abs_time`: [in] The point in time the given function should be scheduled at to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts... : [in]` Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
decltype(auto) friend tag_fallback_invoke(async_execute_at_t, Executor &&exec,
                                         hpx::chrono::steady_time_point
                                         const &abs_time, F &&f, Ts&&... ts)
```

```
struct post_after_t : public hpx::functional::detail::tag_fallback<post_after_t>
#include <timed_execution_fwd.hpp> Customization point of asynchronous fire & forget execution agent creation supporting timed execution.
```

This asynchronously (fire & forget) creates a single function invocation f() using the associated executor at the given point in time.

Note This calls exec.post_after(rel_time, f, ts...), if available, otherwise it emulates timed scheduling by delaying calling execution::post() on the underlying non-time-scheduled execution agent.

Parameters

- exec: [in] The executor object to use for scheduling of the function *f*.
- rel_time: [in] The duration of time after which the given function should be scheduled to run.
- f: [in] The function which will be scheduled using the given executor.
- ts...: [in] Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
decltype(auto) friend tagFallback_invoke(post_after_t, Executor &&exec,
                                         hpx::chrono::steady_duration const
                                         &rel_time, F &&f, Ts&&... ts)

struct post_at_t : public hpx::functional::detail::tagFallback<post_at_t>
#include <timed_execution_fwd.hpp> Customization point of asynchronous fire & forget execution agent creation supporting timed execution.
```

This asynchronously (fire & forget) creates a single function invocation f() using the associated executor at the given point in time.

Note This calls exec.post_at(abs_time, f, ts...), if available, otherwise it emulates timed scheduling by delaying calling execution::post() on the underlying non-time-scheduled execution agent.

Parameters

- exec: [in] The executor object to use for scheduling of the function *f*.
- abs_time: [in] The point in time the given function should be scheduled at to run.
- f: [in] The function which will be scheduled using the given executor.
- ts...: [in] Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
decltype(auto) friend tagFallback_invoke(post_at_t, Executor &&exec,
                                         hpx::chrono::steady_time_point
                                         const &abs_time, F &&f, Ts&&...
                                         ts)

struct syncExecuteAfter_t : public hpx::functional::detail::tagFallback<syncExecuteAfter_t>
#include <timed_execution_fwd.hpp> Customization point of synchronous execution agent creation supporting timed execution.
```

This synchronously creates a single function invocation f() using the associated executor at the given point in time.

Return f(ts...)’s result

Note This calls `exec.sync_execute_after(rel_time, f, ts...)`, if available, otherwise it emulates timed scheduling by delaying calling `execution::sync_execute()` on the underlying non-time-scheduled execution agent.

Parameters

- `exec`: [in] The executor object to use for scheduling of the function *f*.
- `rel_time`: [in] The duration of time after which the given function should be scheduled to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts...`: [in] Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
decltype(auto) friend tag_fallback_invoke(sync_execute_after_t, Executor
                                         &&exec, hpx::chrono::steady_duration
                                         const &rel_time, F &&f, Ts&&... ts)

struct sync_execute_at_t : public hpx::functional::detail::tag_fallback<sync_execute_at_t>
#include <timed_execution_fwd.hpp> Customization point of synchronous execution agent creation supporting timed execution.
```

This synchronously creates a single function invocation *f()* using the associated executor at the given point in time.

Return *f(ts...)*'s result

Note This calls `exec.sync_execute_at(abs_time, f, ts...)`, if available, otherwise it emulates timed scheduling by delaying calling `execution::sync_execute()` on the underlying non-time-scheduled execution agent.

Parameters

- `exec`: [in] The executor object to use for scheduling of the function *f*.
- `abs_time`: [in] The point in time the given function should be scheduled at to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts...`: [in] Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
decltype(auto) friend tag_fallback_invoke(sync_execute_at_t, Executor &&exec,
                                         hpx::chrono::steady_time_point
                                         const &abs_time, F &&f, Ts&&... ts)
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

TypeDefs

```
using sequenced_timed_executor = timed_executor<hpx::execution::sequenced_executor>
using parallel_timed_executor = timed_executor<hpx::execution::parallel_executor>

template<typename BaseExecutor>
struct timed_executor
```

Public Types

```
typedef std::decay<BaseExecutor>::type base_executor_type
typedef hpx::traits::executor_execution_category<base_executor_type>::type execution_category
typedef hpx::traits::executor_parameters_type<base_executor_type>::type parameters_type
```

Public Functions

```
timed_executor(hpx::chrono::steady_time_point const &abs_time)
timed_executor(hpx::chrono::steady_duration const &rel_time)

template<typename Executor>
timed_executor(Executor &&exec, hpx::chrono::steady_time_point const &abs_time)

template<typename Executor>
timed_executor(Executor &&exec, hpx::chrono::steady_duration const &rel_time)

template<typename F, typename ...Ts>
hpx::util::detail::invoke_deferred_result<F, Ts...>::type sync_execute(F &&f, Ts&&...
ts)

template<typename F, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute(F
&&f,
Ts&&...
ts)

template<typename F, typename ...Ts>
void post(F &&f, Ts&&... ts)
```

Public Members

```
BaseExecutor exec_
std::chrono::steady_clock::time_point execute_at_
namespace hpx

namespace parallel

namespace execution
```

TypeDefs

```
template<typename T>
using is_timed_executor_t = typename is_timed_executor<T>::type
```

Variables

```
template<typename T>
constexpr bool is_timed_executor_v = is_timed_executor<T>::value
```

timing

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace chrono

struct high_resolution_clock
```

Public Static Functions

```
static std::uint64_t now()
static constexpr std::uint64_t() hpx::chrono::high_resolution_clock::min()
static constexpr std::uint64_t() hpx::chrono::high_resolution_clock::max()

namespace util
```

TypeDefs

```
typedef hpx::chrono::steady_duration instead

namespace hpx
```

```
namespace chrono

class high_resolution_timer
```

Public Functions

```
high_resolution_timer()
constexpr high_resolution_timer(double t)
void restart()
double elapsed() const
```

```
std::int64_t elapsed_microseconds() const  
std::int64_t elapsed_nanoseconds() const
```

Public Static Functions

```
static double now()  
static constexpr double elapsed_max()  
static constexpr double elapsed_min()
```

Protected Static Functions

```
static std::uint64_t take_time_stamp()
```

Private Members

```
std::uint64_t start_time_  
namespace hpx  
namespace util
```

```
template<typename T>  
struct scoped_timer
```

Public Functions

```
scoped_timer(T &t, bool enabled = true)  
scoped_timer(scoped_timer const&)  
scoped_timer(scoped_timer &&rhs)  
~scoped_timer()  
scoped_timer &operator=(scoped_timer const &rhs)  
scoped_timer &operator=(scoped_timer &&rhs)  
bool enabled() const
```

Private Members

```
std::uint64_t started_at_
T *t_
```

```
namespace hpx
```

```
namespace chrono
```

```
class steady_duration
```

Public Functions

```
steady_duration(value_type const &rel_time)
```

```
template<typename Rep, typename Period>
```

```
steady_duration(std::chrono::duration<Rep, Period> const &rel_time)
```

```
value_type const &value() const
```

```
steady_clock::time_point from_now() const
```

Private Types

```
typedef steady_clock::duration value_type
```

Private Members

```
value_type _rel_time
```

```
class steady_time_point
```

Public Functions

```
steady_time_point(value_type const &abs_time)
```

```
template<typename Clock, typename Duration>
```

```
steady_time_point(std::chrono::time_point<Clock, Duration> const &abs_time)
```

```
value_type const &value() const
```

Private Types

```
typedef steady_clock::time_point value_type
```

Private Members

```
value_type _abs_time  
namespace hpx
```

```
namespace util
```

```
class tick_counter
```

Public Functions

```
tick_counter(std::uint64_t &output)  
~tick_counter()
```

Protected Static Functions

```
static std::uint64_t take_time_stamp()
```

Private Members

```
const std::uint64_t start_time_  
std::uint64_t &output_
```

topology

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace threads
```

Typedefs

```
using hwloc_bitmap_ptr = std::shared_ptr<hpx_hwloc_bitmap_wrapper>
```

Enums

```
enum hpx_hwloc_membind_policy
```

Please see hwloc documentation for the corresponding enums HWLOC_MEMBIND_XXX.

Values:

```
membind_default = HWLOC_MEMBIND_DEFAULT
```

```
membind_firsttouch = HWLOC_MEMBIND_FIRSTTOUCH
```

```
membind_bind = HWLOC_MEMBIND_BIND
```

```
membind_interleave = HWLOC_MEMBIND_INTERLEAVE
membind_replicate = HWLOC_MEMBIND_REPLICATE
membind_nexttouch = HWLOC_MEMBIND_NEXTTOUCH
membind_mixed = HWLOC_MEMBIND_MIXED
membind_user = HWLOC_MEMBIND_MIXED + 256
```

Functions

```
topology &create_topology()
HPX_NODISCARD unsigned int hpx::threads::hardware_concurrency()

std::size_t get_memory_page_size()

struct hpx_hwloc_bitmap_wrapper
```

Public Functions

```
HPX_NON_COPYABLE(hpx_hwloc_bitmap_wrapper)

hpx_hwloc_bitmap_wrapper()
hpx_hwloc_bitmap_wrapper(void *bmp)
~hpx_hwloc_bitmap_wrapper()
void reset(hwloc_bitmap_t bmp)
operator bool() const
hwloc_bitmap_t get_bmp() const
```

Private Members

```
hwloc_bitmap_t bmp_
```

Friends

```
std::ostream &operator<<(std::ostream &os, hpx_hwloc_bitmap_wrapper const *bmp)

struct topology
```

Public Functions

`topology()`

`~topology()`

`std::size_t get_socket_number(std::size_t num_thread, error_code& = throws) const`

Return the Socket number of the processing unit the given thread is running on.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`std::size_t get numa_node_number(std::size_t num_thread, error_code& = throws)`

`const`

Return the NUMA node number of the processing unit the given thread is running on.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_cref_type get_machine_affinity_mask(error_code &ec = throws) const`

Return a bit mask where each set bit corresponds to a processing unit available to the application.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_type get_service_affinity_mask(mask_cref_type used_processing_units, error_code &ec = throws)`

`const`

Return a bit mask where each set bit corresponds to a processing unit available to the service threads in the application.

Parameters

- `used_processing_units`: [in] This is the mask of processing units which are not available for service threads.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_cref_type get_socket_affinity_mask(std::size_t num_thread, error_code &ec = throws)`

`const`

Return a bit mask where each set bit corresponds to a processing unit available to the given thread inside the socket it is running on.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_cref_type get numa_node_affinity_mask(std::size_t num_thread, error_code &ec = throws)`

`const`

Return a bit mask where each set bit corresponds to a processing unit available to the given thread inside the NUMA domain it is running on.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_type get_numa_node_affinity_mask_from_numa_node (std::size_t num_node) const`
 Return a bit mask where each set bit corresponds to a processing unit associated with the given NUMA node.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_cref_type get_core_affinity_mask (std::size_t num_thread, error_code &ec = throws) const`
 Return a bit mask where each set bit corresponds to a processing unit available to the given thread inside the core it is running on.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_cref_type get_thread_affinity_mask (std::size_t num_thread, error_code &ec = throws) const`
 Return a bit mask where each set bit corresponds to a processing unit available to the given thread.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`void set_thread_affinity_mask (mask_cref_type mask, error_code &ec = throws) const`
 Use the given bit mask to set the affinity of the given thread. Each set bit corresponds to a processing unit the thread will be allowed to run on.

Note Use this function on systems where the affinity must be set from inside the thread itself.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`mask_type get_thread_affinity_mask_from_lva (void const *lva, error_code &ec = throws) const`
 Return a bit mask where each set bit corresponds to a processing unit co-located with the memory the given address is currently allocated on.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`void print_affinity_mask (std::ostream &os, std::size_t num_thread, mask_cref_type m, const std::string &pool_name) const`

Prints the.

Parameters

- `m`: to `os` in a human readable form

```
bool reduce_thread_priority(error_code &ec = throws) const
    Reduce thread priority of the current thread.
```

Parameters

- ec: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
std::size_t get_number_of_sockets() const
    Return the number of available NUMA domains.
```

```
std::size_t get_number_of numa_nodes() const
    Return the number of available NUMA domains.
```

```
std::size_t get_number_of_cores() const
    Return the number of available cores.
```

```
std::size_t get_number_of_pus() const
    Return the number of available hardware processing units.
```

```
std::size_t get_number_of numa_node_cores(std::size_t numa) const
    Return number of cores in given numa domain.
```

```
std::size_t get_number_of numa_node_pus(std::size_t numa) const
    Return number of processing units in a given numa domain.
```

```
std::size_t get_number_of_socket_pus(std::size_t socket) const
    Return number of processing units in a given socket.
```

```
std::size_t get_number_of_core_pus(std::size_t core) const
    Return number of processing units in given core.
```

```
std::size_t get_number_of_socket_cores(std::size_t socket) const
    Return number of cores units in given socket.
```

```
std::size_t get_core_number(std::size_t num_thread, error_code& = throws) const
```

```
std::size_t get_pu_number(std::size_t num_core, std::size_t num_pu, error_code &ec = throws) const
```

```
mask_type get_cpubind_mask(error_code &ec = throws) const
```

```
mask_type get_cpubind_mask(std::thread &handle, error_code &ec = throws) const
```

```
hwloc_bitmap_ptr cpuset_to_nodeset(mask_cref_type cpuset) const
    convert a cpu mask into a numa node mask in hwloc bitmap form
```

```
void write_to_log() const
```

```
void *allocate(std::size_t len) const
```

This is equivalent to malloc(), except that it tries to allocate page-aligned memory from the OS.

```
void *allocate_membind(std::size_t len, hwloc_bitmap_ptr bitmap,
                        hpx_hwloc_membind_policy policy, int flags) const
```

allocate memory with binding to a numa node set as specified by the policy and flags (see hwloc docs)

```
threads::mask_type get_area_membind_nodeset(const void *addr, std::size_t len)
                                                const
```

```

bool set_area_membind_nodeset(const void *addr, std::size_t len, void *nodeset)
                                const

int get_numa_domain(const void *addr) const

void deallocate(void *addr, std::size_t len) const
    Free memory that was previously allocated by allocate.

void print_vector(std::ostream &os, std::vector<std::size_t> const &v) const

void print_mask_vector(std::ostream &os, std::vector<mask_type> const &v) const

void print_hwloc(std::ostream&) const

mask_type init_socket_affinity_mask_from_socket(std::size_t num_socket)
                                                const

mask_type init numa_node_affinity_mask_from numa_node(std::size_t num_numa_node)
                                                const

mask_type init_core_affinity_mask_from_core(std::size_t num_core,
                                              mask_cref_type default_mask
                                              = empty_mask) const

mask_type init_thread_affinity_mask(std::size_t num_thread) const

mask_type init_thread_affinity_mask(std::size_t num_core, std::size_t num_pu)
                                                const

hwloc_bitmap_t mask_to_bitmap(mask_cref_type mask, hwloc_obj_type_t htype) const

mask_type bitmap_to_mask(hwloc_bitmap_t bitmap, hwloc_obj_type_t htype) const

```

Private Types

```
using mutex_type = hpx::util::spinlock
```

Private Functions

```

std::size_t init_node_number(std::size_t num_thread, hwloc_obj_type_t type)

std::size_t init_socket_number(std::size_t num_thread)

std::size_t init numa_node_number(std::size_t num_thread)

std::size_t init_core_number(std::size_t num_thread)

void extract_node_mask(hwloc_obj_t parent, mask_type &mask) const

std::size_t extract_node_count(hwloc_obj_t parent, hwloc_obj_type_t type, std::size_t count) const

mask_type init_machine_affinity_mask() const

mask_type init_socket_affinity_mask(std::size_t num_thread) const

mask_type init numa_node_affinity_mask(std::size_t num_thread) const

mask_type init_core_affinity_mask(std::size_t num_thread) const

```

```
void init_num_of_pus()
```

Private Members

```
hwloc_topology_t topo
std::size_t num_of_pus_
bool use_pus_as_cores_
mutex_type topo_mtx
std::vector<std::size_t> socket_numbers_
std::vector<std::size_t> numa_node_numbers_
std::vector<std::size_t> core_numbers_
mask_type machine_affinity_mask_
std::vector<mask_type> socket_affinity_masks_
std::vector<mask_type> numa_node_affinity_masks_
std::vector<mask_type> core_affinity_masks_
std::vector<mask_type> thread_affinity_masks_
```

Private Static Attributes

```
mask_type empty_mask
std::size_t memory_page_size_
constexpr std::size_t pu_offset = 0
constexpr std::size_t core_offset = 0
```

Friends

```
std::size_t get_memory_page_size()
```

type_support

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

Typedefs

```
template<typename ...T>
using always_void_t = typename always_void<T...>::type

template<typename ...T>
struct always_void
```

Public Types

```
template<>
using type = void

namespace hpx

namespace util
```

Typedefs

```
template<typename T>
using decay_unwrap_t = typename decay_unwrap<T>::type

namespace hpx
```

```
namespace util
```

Typedefs

```
template<template<typename...> class Op, typename ...Args>
using is_detected = typename detail::detector<nonesuch, void, Op, Args...>::value_t

template<template<typename...> class Op, typename ...Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type

template<typename Default, template<typename...> class Op, typename ...Args>
using detected_or = detail::detector<Default, void, Op, Args...>

template<typename Default, template<typename...> class Op, typename ...Args>
using detected_or_t = typename detected_or<Default, Op, Args...>::type

template<typename Expected, template<typename...> class Op, typename ...Args>
using is_detected_exact = std::is_same<Expected, detected_t<Op, Args...>>

template<typename To, template<typename...> class Op, typename ...Args>
using is_detected_convertible = std::is_convertible<detected_t<Op, Args...>, To>

struct nonesuch
```

Public Functions

```
nonesuch()  
~nonesuch()  
nonesuch(nonesuch const&)  
void operator=(nonesuch const&)  
namespace hpx
```

```
namespace util
```

```
struct empty_function
```

Public Functions

```
template<typename ...Ts>  
constexpr void operator()(Ts&&...)
```

```
namespace hpx
```

```
namespace util
```

```
template<typename T>  
struct identity
```

Public Types

```
template<>  
using type = T  
namespace hpx
```

```
namespace util
```

Typedefs

```
template<bool Enable, typename C1, typename C2>  
using lazy_conditional_t = typename lazy_conditional<Enable, C1, C2>::type  
namespace hpx
```

```
namespace util
```

```
template<typename T>  
struct lazy_enable_if<true, T>
```

Public Types

```
template<>
using type = typename T::type

namespace hpx
```

```
namespace util
```

Typedefs

```
template<std::size_t... Is>
using index_pack = pack_c<std::size_t, Is...>

template<std::size_t N>
using make_index_pack_t = typename make_index_pack<N>::type

template<std::size_t I, typename ...Ts>
using at_index_t = typename at_index<I, Ts...>::type
```

Variables

```
template<typename ...Ts>
constexpr bool all_of_v = all_of<Ts...>::value

template<typename ...Ts>
constexpr bool any_of_v = any_of<Ts...>::value

template<typename ...Ts>
constexpr bool none_of_v = none_of<Ts...>::value

template<typename ...Ts>
struct pack
```

Public Types

```
typedef pack type
```

Public Static Attributes

```
constexpr std::size_t size = sizeof...(Ts)

template<typename T, T... Vs>
struct pack_c
    Subclassed by hpx::util::detail::make_index_pack_join< index_pack< Left... >, index_pack< Right... >, hpx::util::make_index_pack< 1 >
```

Public Types

```
typedef pack_c type
```

Public Static Attributes

```
constexpr std::size_t size = sizeof...(Vs)
```

Defines

```
HPX_EXPORT_STATIC_
```

```
namespace hpx
```

```
namespace util
```

```
template<typename T, typename Tag = T>
struct static_
```

Public Types

```
typedef T value_type
```

```
typedef T &reference
```

```
typedef T const &const_reference
```

Public Functions

```
HPX_NON_COPYABLE(static_)
```

```
static_()
```

```
operator reference()
```

```
operator const_reference() const
```

```
reference get()
```

```
const_reference get() const
```

Private Types

```
typedef std::add_pointer<value_type>::type pointer
```

```
typedef std::aligned_storage<sizeof(value_type), std::alignment_of<value_type>::value>::type storage_type
```

Private Static Functions

```
static pointer get_address()
```

Private Static Attributes

```
static_<T, Tag>::storage_type data_
std::once_flag constructed_
struct default_constructor
```

Public Static Functions

```
template<>
static void construct()
struct destructor
```

Public Functions

```
template<>
~destructor()
```

Defines

```
HPX_UNUSED(x)
```

```
HPX_MAYBE_UNUSED
```

```
namespace hpx
```

```
namespace util
```

Variables

```
constexpr unused_type unused = unused_type()
struct unused_type
```

Public Functions

```
constexpr unused_type()
constexpr unused_type(unused_type const&)
constexpr unused_type(unused_type&&)
template<typename T>
constexpr unused_type(T const&)
```

```
template<typename T>
constexpr unused_type const &operator=(T const&) const

template<typename T>
unused_type &operator=(T const&)

constexpr unused_type const &operator=(unused_type const&) const

constexpr unused_type const &operator=(unused_type&&) const

unused_type &operator=(unused_type const&)

unused_type &operator=(unused_type&&)

template<typename T>
struct unwrap_reference<std::reference_wrapper<T>>
```

Public Types

```
typedef T type

template<typename T>
struct unwrap_reference<std::reference_wrapper<T> const>
```

Public Types

```
typedef T type

namespace hpx

namespace util
```

Functions

```
template<typename T>
unwrap_reference<T>::type &unwrap_ref(T &t)

template<typename T>
struct unwrap_reference
```

Public Types

```
typedef T type

template<typename T>
struct unwrap_reference<std::reference_wrapper<T>>
```

Public Types

```
typedef T type
template<typename T>
struct unwrap_reference<std::reference_wrapper<T> const>
```

Public Types

```
typedef T type
namespace hpx
```

```
namespace util
```

```
template<>
struct void_guard<void>
```

Public Functions

```
template<typename T>
void operator, (T const&) const
```

util

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

Functions

```
std::size_t calculate_fanout (std::size_t size, std::size_t local_fanout)
```

```
namespace hpx
```

```
namespace util
```

Functions

```
std::uint64_t get_and_reset_value (std::uint64_t &value, bool reset)
```

```
std::int64_t get_and_reset_value (std::int64_t &value, bool reset)
```

```
template<typename T>
T get_and_reset_value (std::atomic<T> &value, bool reset)
```

```
std::vector<std::int64_t> get_and_reset_value (std::vector<std::int64_t> &value, bool reset)
```

```
namespace hpx
```

```
    namespace util
```

Functions

```
template<typename DestType, typename Config, typename std::enable_if<!std::is_same<DestType, std::string>::value>
DestType get_entry_as(Config const &config, std::string const &key, DestType const &dflt)
```

```
namespace hpx
```

```
    namespace util
```

Functions

```
template<typename Iterator>
bool insert_checked(std::pair<Iterator, bool> const &r)
```

Helper function for writing predicates that test whether an std::map insertion succeeded. This inline template function negates the need to explicitly write the sometimes lengthy std::pair<Iterator, bool> type.

Return This function returns **r.second**.

Parameters

- **r**: [in] The return value of a std::map insert operation.

```
template<typename Iterator>
```

```
bool insert_checked(std::pair<Iterator, bool> const &r, Iterator &it)
```

Helper function for writing predicates that test whether an std::map insertion succeeded. This inline template function negates the need to explicitly write the sometimes lengthy std::pair<Iterator, bool> type.

Return This function returns **r.second**.

Parameters

- **r**: [in] The return value of a std::map insert operation.
- **r**: [out] A reference to an Iterator, which is set to **r.first**.

```
namespace hpx
```

```
    namespace util
```

```
        class ios_flags_saver
```

Public Types

```
typedef ::std::ios_base state_type
typedef ::std::ios_base::fmtflags aspect_type
```

Public Functions

```
ios_flags_saver (state_type &s)
ios_flags_saver (state_type &s, aspect_type const &a)
~ios_flags_saver ()
ios_flags_saver (ios_flags_saver const&)
ios_flags_saver &operator= (ios_flags_saver const&)
void restore ()
```

Private Members

```
state_type &s_save_
const aspect_type a_save_

namespace hpx

namespace util

struct manage_config
```

Public Types

```
typedef std::map<std::string, std::string> map_type
```

Public Functions

```
manage_config (std::vector<std::string> const &cfg)
void add (std::vector<std::string> const &cfg)
template<typename T>
T get_value (std::string const &key, T dflt = T()) const
```

Public Members

```
map_type config_
namespace hpx
namespace util
```

Functions

```
std::string regex_from_pattern(std::string const &pattern, error_code &ec = throws)
namespace hpx
namespace util
```

Functions

```
bool parse_sed_expression(std::string const &input, std::string &search, std::string &replace)
```

Parse a sed command.

Return *true* if the parsing was successful, false otherwise.

Note Currently, only supports search and replace syntax (s/search/replace/)

Parameters

- **input:** [in] The content to parse.
- **search:** [out] If the parsing is successful, this string is set to the search expression.
- **search:** [out] If the parsing is successful, this string is set to the replace expression.

struct sed_transform

#include <sed_transform.hpp> An unary function object which applies a sed command to its subject and returns the resulting string.

Note Currently, only supports search and replace syntax (s/search/replace/)

Public Functions

```
sed_transform(std::string const &search, std::string const &replace)
sed_transform(std::string const &expression)
std::string operator()(std::string const &input) const
operator bool() const
bool operator!() const
```

Private Members

std::shared_ptr<command> **command_**

version

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace hpx

Functions

```
std::uint8_t major_version()  
std::uint8_t minor_version()  
std::uint8_t subminor_version()  
std::uint32_t full_version()  
std::string full_version_as_string()  
std::uint8_t agas_version()  
std::string tag()  
std::string full_build_string()  
std::string build_string()  
std::string boost_version()  
std::string boost_platform()  
std::string boost_compiler()  
std::string boost_stdlib()  
std::string copyright()  
std::string complete_version()  
std::string build_type()  
std::string build_date_time()  
std::string configuration_string()
```

actions

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
namespace hpx
```

Functions

```
bool is_pre_startup()
```

actions_base

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
namespace hpx
```

```
namespace actions
```

Functions

```
template<typename Action>
constexpr threads::thread_priority action_priority()
```

Defines

```
HPX_REGISTER_ACTION_DECLARATION(...)
```

Declare the necessary component action boilerplate code.

The macro *HPX_REGISTER_ACTION_DECLARATION* can be used to declare all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to declare the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

```
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server
        : public hpx::components::component_base<server>
    {
        void print_greeting ()
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }

        // Component actions need to be declared, this also defines the
        // type 'print_greeting_action' representing the action.
    };
}
```

(continues on next page)

(continued from previous page)

```

    HPX_DEFINE_COMPONENT_ACTION(server,
        print_greeting, print_greeting_action);
    };
}

// Declare boilerplate code required for each of the component actions.
HPX_REGISTER_ACTION_DECLARATION(app::server::print_greeting_action)

```

Example:

Note This macro has to be used once for each of the component actions defined using one of the `HPX_DEFINE_COMPONENT_ACTION` macros. It has to be visible in all translation units using the action, thus it is recommended to place it into the header file defining the component.

`HPX_REGISTER_ACTION_DECLARATION(...)`

`HPX_REGISTER_ACTION_DECLARATION_1(action)`

`HPX_REGISTER_ACTION(...)`

Define the necessary component action boilerplate code.

The macro `HPX_REGISTER_ACTION` can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

Note This macro has to be used once for each of the component actions defined using one of the `HPX_DEFINE_COMPONENT_ACTION` or `HPX_DEFINE_PLAIN_ACTION` macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note Only one of the forms of this macro `HPX_REGISTER_ACTION` or `HPX_REGISTER_ACTION_ID` should be used for a particular action, never both.

`HPX_REGISTER_ACTION_ID(action, actionname, actionid)`

Define the necessary component action boilerplate code and assign a predefined unique id to the action.

The macro `HPX_REGISTER_ACTION` can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

The parameter *actionname* specifies an unique name of the action to be used for serialization purposes. The second parameter has to be usable as a plain (non-qualified) C++ identifier, it should not contain special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

The parameter *actionid* specifies an unique integer value which will be used to represent the action during serialization.

Note This macro has to be used once for each of the component actions defined using one of the `HPX_DEFINE_COMPONENT_ACTION` or global actions `HPX_DEFINE_PLAIN_ACTION` macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note Only one of the forms of this macro `HPX_REGISTER_ACTION` or `HPX_REGISTER_ACTION_ID` should be used for a particular action, never both.

```
namespace hpx
```

```
namespace actions
```

```
template<typename Component, typename Signature, typename Derived>
struct basic_action
    #include <basic_action_fwd.hpp>
Template Parameters

- Component: component type
- Signature: return type and arguments
- Derived: derived action class

```

Defines

`HPX_DEFINE_COMPONENT_ACTION(...)`

Registers a member function of a component as an action type with HPX.

The macro `HPX_DEFINE_COMPONENT_ACTION` can be used to register a member function of a component as an action type named `action_type`.

The parameter `component` is the type of the component exposing the member function `func` which should be associated with the newly defined action type. The parameter `action_type` is the name of the action type to register with HPX.

```
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server
        : public hpx::components::component_base<server>
    {
        void print_greeting() const
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }

        // Component actions need to be declared, this also defines the
        // type 'print_greeting_action' representing the action.
        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.

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.

```
namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename Action>
using action_continuation_t = typename action_continuation<Action>::type

template<typename Action, typename Enable = void>
struct action_continuation
```

Public Types

```
template<>
using type = hpx::actions::typed_continuation<typename Action::local_result_type, typename Action::remote
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename Action, typename Enable = void>
struct action_decorate_continuation
```

Public Types

```
template<>
using continuation_type = typename hpx::traits::action_continuation<Action>::type
```

Public Static Functions

```
static constexpr bool call(continuation_type&)
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename Action, typename Enable = void>
struct action_does_termination_detection
```

Public Static Functions

```
static constexpr bool call()

namespace hpx

namespace traits

template<typename Action, typename Enable = void>
struct action_is_target_valid
```

Public Static Functions

```
static bool call(naming::id_type const &id)

namespace hpx
```

```
namespace traits
```

Variables

```
template<typename Action>
constexpr threads::thread_priority action_priority_v = action_priority<Action>::value

template<typename Action, typename Enable = void>
struct action_priority
```

Public Static Attributes

```
constexpr threads::thread_priority value = threads::thread_priority::default_

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename Result>
using action_remote_result_t = typename action_remote_result<Result>::type

namespace hpx
```

```
namespace traits
```

```
template<typename Action, typename Enable = void>
struct action_schedule_thread
```

Public Static Functions

```
static void call (naming::address_type lva, naming::component_type comptype,
threads::thread_init_data &data)  
namespace hpx
```

```
namespace traits
```

```
template<typename Action, typename Enable = void>  
struct action_select_direct_execution
```

Public Static Functions

```
static constexpr launch call (launch policy, naming::address_type lva)  
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename Action>  
constexpr threads::thread_stacksize action_stacksize_v = action_stacksize<Action>::value  
template<typename Action, typename Enable = void>  
struct action_stacksize
```

Public Static Attributes

```
constexpr threads::thread_stacksize value = threads::thread_stacksize::default_  
namespace hpx
```

```
namespace traits
```

```
template<typename Continuation, typename Enable = void>  
struct action_trigger_continuation
```

Public Static Functions

```
template<typename F, typename ...Ts>  
static void call (Continuation&&, F&&, Ts&&...>)  
namespace hpx
```

```
namespace traits
```

```
template<typename Action, typename Enable = void>
struct action_was_object_migrated
```

Public Static Functions

```
static std::pair<bool, components::pinned_ptr> call(hpx::naming::gid_type const &id,
                                                       naming::address_type lva)
static std::pair<bool, components::pinned_ptr> call(hpx::naming::id_type const &id,
                                                       naming::address_type lva)
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename Action, typename Enable = void>
struct extract_action
```

Public Types

```
template<>
using type = typename Action::derived_type
template<>
using result_type = typename type::result_type
template<>
using local_result_type = typename type::local_result_type
template<>
using remote_result_type = typename type::remote_result_type
```

```
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_client_v = is_client<T>::value
template<typename T>
constexpr bool is_client_or_client_array_v = is_client_or_client_array<T>::value
```

```
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename Continuation>
constexpr bool is_continuation_v = is_continuation<Continuation>::value

namespace hpx
```

```
namespace traits
```

Variables

```
template<typename T>
constexpr bool is_distribution_policy_v = is_distribution_policy<T>::value

template<typename Policy, typename Enable = void>
struct num_container_partitions
```

Public Static Functions

```
static std::size_t call(Policy const &policy)
```

agas

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace agas
```

```
struct addressing_service
```

Public Types

```
using component_id_type = components::component_type
using iterate_names_return_type = std::map<std::string, naming::id_type>
using iterate_types_function_type = hpx::util::function<void (std::string
                           const&, components::component_type)>
using mutex_type = hpx::lcos::local::spinlock
using gva_cache_type = hpx::util::cache::lru_cache<gva_cache_key, gva, hpx::util::cache::statistics::local_full_
using migrated_objects_table_type = std::set<naming::gid_type>
using refcnt_requests_type = std::map<naming::gid_type, std::int64_t>
using resolved_localities_type = std::map<naming::gid_type, parcelset::endpoints_type>
```

Public Functions

```
HPX_NON_COPYABLE (addressing_service)

addressing_service (util::runtime_configuration const &ini_)

~addressing_service ()

void bootstrap (parcelset::endpoints_type const &endpoints, util::runtime_configuration
&rtcfg)

void initialize (std::uint64_t rts_lva)

void adjust_local_cache_size (std::size_t)
    Adjust the size of the local AGAS Address resolution cache.

state get_status () const

void set_status (state new_state)

naming::gid_type const &get_local_locality (error_code& = throws) const

void set_local_locality (naming::gid_type const &g)

void register_console (parcelset::endpoints_type const &eps)

bool is_bootstrap () const

bool is_console () const
    Returns whether this addressing_service represents the console locality.

bool is_connecting () const
    Returns whether this addressing_service is connecting to a running application.

bool resolve_locally_known_addresses (naming::gid_type const &id, naming::address &addr)

void register_server_instances ()

void garbage_collect_non_blocking (error_code &ec = throws)

void garbage_collect (error_code &ec = throws)

std::int64_t synchronize_with_async_incref (hpx::future<std::int64_t> fut, naming::id_type const &id, std::int64_t compensated_credit)

server::primary_namespace &get_local_primary_namespace_service ()

naming::address::address_type get_primary_ns_lva () const

naming::address::address_type get_symbol_ns_lva () const

server::component_namespace *get_local_component_namespace_service ()

server::locality_namespace *get_local_locality_namespace_service ()

server::symbol_namespace &get_local_symbol_namespace_service ()

naming::address::address_type get_runtime_support_lva () const
```

```

std::uint64_t get_cache_entries (bool)
std::uint64_t get_cache_hits (bool)
std::uint64_t get_cache_misses (bool)
std::uint64_t get_cache_evictions (bool)
std::uint64_t get_cache_insertions (bool)
std::uint64_t get_cache_get_entry_count (bool reset)
std::uint64_t get_cache_insertion_entry_count (bool reset)
std::uint64_t get_cache_update_entry_count (bool reset)
std::uint64_t get_cache_erase_entry_count (bool reset)
std::uint64_t get_cache_get_entry_time (bool reset)
std::uint64_t get_cache_insertion_entry_time (bool reset)
std::uint64_t get_cache_update_entry_time (bool reset)
std::uint64_t get_cache_erase_entry_time (bool reset)
bool register_locality (parcelset::endpoints_type const &endpoints, naming::gid_type &prefix, std::uint32_t num_threads, error_code &ec = throws)
    Add a locality to the runtime.

parcelset::endpoints_type const &resolve_locality (naming::gid_type const &gid, error_code &ec = throws)
    Resolve a locality to its prefix.

```

Return Returns an empty vector if the locality is not registered.

```

bool has_resolved_locality (naming::gid_type const &gid)
bool unregister_locality (naming::gid_type const &gid, error_code &ec = throws)
    Remove a locality from the runtime.

void remove_resolved_locality (naming::gid_type const &gid)
    remove given locality from locality cache

bool get_console_locality (naming::gid_type &locality, error_code &ec = throws)
    Get locality locality_id of the console locality.

```

Return This function returns *true* if a console locality_id exists and returns *false* otherwise.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *locality_id*: [out] The locality_id value uniquely identifying the console locality. This is valid only, if the return value of this function is true.
- *try_cache*: [in] If this is set to true the console is first tried to be found in the local cache. Otherwise this function will always query AGAS, even if the console locality_id is already known locally.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
bool get_localities (std::vector<naming::gid_type> &locality_ids, components::component_type type, error_code &ec = throws)
```

Query for the locality_ids of all known localities.

This function returns the locality_ids of all localities known to the AGAS server or all localities having a registered factory for a given component type.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *locality_ids*: [out] The vector will contain the prefixes of all localities registered with the AGAS server. The returned vector holds the prefixes representing the runtime_support components of these localities.
- *type*: [in] The component type will be used to determine the set of prefixes having a registered factory for this component. The default value for this parameter is *components::component_invalid*, which will return prefixes of all localities.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
bool get_localities (std::vector<naming::gid_type> &locality_ids, error_code &ec = throws)
```

```
hpx::future<std::uint32_t> get_num_localities_async (components::component_type type = components::component_invalid) const
```

Query for the number of all known localities.

This function returns the number of localities known to the AGAS server or the number of localities having a registered factory for a given component type.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *type*: [in] The component type will be used to determine the set of prefixes having a registered factory for this component. The default value for this parameter is *components::component_invalid*, which will return prefixes of all localities.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
std::uint32_t get_num_localities (components::component_type type, error_code &ec = throws) const
```

```
std::uint32_t get_num_localities (error_code &ec = throws) const
```

```
hpx::future<std::uint32_t> get_num_overall_threads_async () const
```

```
std::uint32_t get_num_overall_threads (error_code &ec = throws) const
```

```
hpx::future<std::vector<std::uint32_t>> get_num_threads_async () const
```

```
std::vector<std::uint32_t> get_num_threads (error_code &ec = throws) const
```

```
components::component_type get_component_id (std::string const &name, error_code &ec = throws)
```

Return a unique id usable as a component type.

This function returns the component type id associated with the given component name. If this is the first request for this component name a new unique id will be created.

Return The function returns the currently associated component type. Any error results in an exception thrown from this function.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *name*: [in] The component name (string) to get the component type for.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
void iterate_types (iterate_types_function_type const &f, error_code &ec = throws)
std::string get_component_type_name (components::component_type id, error_code &ec
= throws)
components::component_type register_factory (naming::gid_type const &locality_id,
                                             std::string const &name, error_code
                                             &ec = throws)
```

Register a factory for a specific component type.

This function allows to register a component factory for a given locality and component type.

Return The function returns the currently associated component type. Any error results in an exception thrown from this function. The returned component type is the same as if the function *get_component_id* was called using the same component name.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *locality_id*: [in] The locality value uniquely identifying the given locality the factory needs to be registered for.
- *name*: [in] The component name (string) to register a factory for the given component type for.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
components::component_type register_factory (std::uint32_t locality_id, std::string
                                            const &name, error_code &ec =
                                            throws)
```

```
bool get_id_range (std::uint64_t count, naming::gid_type &lower_bound, naming::gid_type
                   &upper_bound, error_code &ec = throws)
```

Get unique range of freely assignable global ids.

Every locality needs to be able to assign global ids to different components without having to consult the AGAS server for every id to generate. This function can be called to preallocate a range of ids usable for this purpose.

Return This function returns *true* if a new range has been generated (it has been called for the first time for the given locality) and returns *false* if this locality already got a range assigned in an earlier call. Any error results in an exception thrown from this function.

Note This function assigns a range of global ids usable by the given locality for newly created components. Any of the returned global ids still has to be bound to a local address, either by calling *bind* or *bind_range*.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *l*: [in] The locality the locality id needs to be generated for. Repeating calls using the same locality results in identical locality_id values.
- *count*: [in] The number of global ids to be generated.
- *lower_bound*: [out] The lower bound of the assigned id range. The returned value can be used as the first id to assign. This is valid only, if the return value of this function is true.
- *upper_bound*: [out] The upper bound of the assigned id range. The returned value can be used as the last id to assign. This is valid only, if the return value of this function is true.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
bool bind_local(naming::gid_type const &id, naming::address const &addr, error_code  
          &ec = throws)
```

Bind a global address to a local address.

Every element in the HPX namespace has a unique global address (global id). This global address has to be associated with a concrete local address to be able to address an instance of a component using its global address.

Return This function returns *true*, if this global id got associated with an local address. It returns *false* otherwise.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note Binding a gid to a local address sets its global reference count to one.

Parameters

- *id*: [in] The global address which has to be bound to the local address.
- *addr*: [in] The local address to be bound to the global address.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
hpx::future<bool> bind_async(naming::gid_type const &id, naming::address const  
                           &addr, std::uint32_t locality_id)
```

```
hpx::future<bool> bind_async(naming::gid_type const &id, naming::address const  
                           &addr, naming::gid_type const &locality)
```

```
bool bind_range_local(naming::gid_type const &lower_id, std::uint64_t count, naming::address const &baseaddr, std::uint64_t offset, error_code  
                      &ec = throws)
```

Bind unique range of global ids to given base address.

Every locality needs to be able to bind global ids to different components without having to consult the AGAS server for every id to bind. This function can be called to bind a range of consecutive global ids to a range of consecutive local addresses (separated by a given *offset*).

Return This function returns *true*, if the given range was successfully bound. It returns *false* otherwise.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note Binding a gid to a local address sets its global reference count to one.

Parameters

- *lower_id*: [in] The lower bound of the assigned id range. The value can be used as the first id to assign.
- *count*: [in] The number of consecutive global ids to bind starting at *lower_id*.

- `baseaddr`: [in] The local address to bind to the global id given by `lower_id`. This is the base address for all additional local addresses to bind to the remaining global ids.
- `offset`: [in] The offset to use to calculate the local addresses to be bound to the range of global ids.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
hpx::future<bool> bind_range_async(naming::gid_type const &lower_id, std::uint64_t
                                     count, naming::address const &baseaddr,
                                     std::uint64_t offset, naming::gid_type const
                                     &locality)
```

```
hpx::future<bool> bind_range_async(naming::gid_type const &lower_id, std::uint64_t
                                     count, naming::address const &baseaddr,
                                     std::uint64_t offset, std::uint32_t locality_id)
```

`bool unbind_local(naming::gid_type const &id, error_code &ec = throws)`
Unbind a global address.

Note Remove the association of the given global address with any local address, which was bound to this global address. Additionally it returns the local address which was bound at the time of this call.

Return The function returns `true` if the association has been removed, and it returns `false` if no association existed. Any error results in an exception thrown from this function.

Note You can unbind only global ids bound using the function `bind`. Do not use this function to unbind any of the global ids bound using `bind_range`.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note This function will raise an error if the global reference count of the given gid is not zero!
TODO: confirm that this happens.

Parameters

- `id`: [in] The global address (id) for which the association has to be removed.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`bool unbind_local(naming::gid_type const &id, naming::address &addr, error_code &ec = throws)`
Unbind a global address.

Remove the association of the given global address with any local address, which was bound to this global address. Additionally it returns the local address which was bound at the time of this call.

Return The function returns `true` if the association has been removed, and it returns `false` if no association existed. Any error results in an exception thrown from this function.

Note You can unbind only global ids bound using the function `bind`. Do not use this function to unbind any of the global ids bound using `bind_range`.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note This function will raise an error if the global reference count of the given gid is not zero!
TODO: confirm that this happens.

Parameters

- `id`: [in] The global address (id) for which the association has to be removed.
- `addr`: [out] The local address which was associated with the given global address (id). This is valid only if the return value of this function is true.

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
bool unbind_range_local(naming::gid_type const &lower_id, std::uint64_t count, error_code &ec = throws)  
Unbind the given range of global ids.
```

Return This function returns *true* if a new range has been generated (it has been called for the first time for the given locality) and returns *false* if this locality already got a range assigned in an earlier call. Any error results in an exception thrown from this function.

Note You can unbind only global ids bound using the function `bind_range`. Do not use this function to unbind any of the global ids bound using `bind`.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note This function will raise an error if the global reference count of the given gid is not zero!
TODO: confirm that this happens.

Parameters

- `lower_id`: [in] The lower bound of the assigned id range. The value must be the first id of the range as specified to the corresponding call to `bind_range`.
- `count`: [in] The number of consecutive global ids to unbind starting at `lower_id`. This number must be identical to the number of global ids bound by the corresponding call to `bind_range`
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
bool unbind_range_local(naming::gid_type const &lower_id, std::uint64_t count, naming::address &addr, error_code &ec = throws)  
Unbind the given range of global ids.
```

Return This function returns *true* if a new range has been generated (it has been called for the first time for the given locality) and returns *false* if this locality already got a range assigned in an earlier call.

Note You can unbind only global ids bound using the function `bind_range`. Do not use this function to unbind any of the global ids bound using `bind`.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note This function will raise an error if the global reference count of the given gid is not zero!

Parameters

- `lower_id`: [in] The lower bound of the assigned id range. The value must be the first id of the range as specified to the corresponding call to `bind_range`.
- `count`: [in] The number of consecutive global ids to unbind starting at `lower_id`. This number must be identical to the number of global ids bound by the corresponding call to `bind_range`
- `addr`: [out] The local address which was associated with the given global address (id). This is valid only if the return value of this function is true.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
hpx::future<naming::address> unbind_range_async(naming::gid_type const  
&lower_id, std::uint64_t count  
= 1)
```

```
bool is_local_address_cached(naming::gid_type const &id, error_code &ec =  
throws)
```

Test whether the given address refers to a local object.

This function will test whether the given address refers to an object living on the locality of the caller.

Return This function returns *true* if the passed address refers to an object which lives on the locality of the caller.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *addr*: [in] The address to test.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
bool is_local_address_cached(naming::gid_type const &id, naming::address &addr,  
error_code &ec = throws)
```

```
bool is_local_lva_encoded_address(std::uint64_t msb)
```

```
bool resolve_local(naming::gid_type const &id, naming::address &addr, error_code  
&ec = throws)
```

Resolve a given global address (*id*) to its associated local address.

This function returns the local address which is currently associated with the given global address (*id*).

Return This function returns *true* if the global address has been resolved successfully (there exists an association to a local address) and the associated local address has been returned. The function returns *false* if no association exists for the given global address. Any error results in an exception thrown from this function.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *id*: [in] The global address (*id*) for which the associated local address should be returned.
- *addr*: [out] The local address which currently is associated with the given global address (*id*), this is valid only if the return value of this function is true.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
bool resolve_local(naming::id_type const &id, naming::address &addr, error_code &ec  
= throws)
```

```
naming::address resolve_local(naming::gid_type const &id, error_code &ec = throws)
```

```
naming::address resolve_local(naming::id_type const &id, error_code &ec = throws)
```

```
hpx::future<naming::address> resolve_async(naming::gid_type const &id)
```

```
hpx::future<naming::address> resolve_async(naming::id_type const &id)
```

```
hpx::future<naming::id_type> get_colocation_id_async(naming::id_type const  
&id)
```

```
bool resolve_full_local(naming::gid_type const &id, naming::address &addr, error_code &ec = throws)
```

```
bool resolve_full_local(naming::id_type const &id, naming::address &addr, error_code &ec = throws)
```

```
naming::address resolve_full_local(naming::gid_type const &id, error_code &ec =  
throws)
```

```
naming::address resolve_full_local (naming::id_type const &id, error_code &ec =  
throws)  
  
hpx::future<naming::address> resolve_full_async (naming::gid_type const &id)  
  
hpx::future<naming::address> resolve_full_async (naming::id_type const &id)  
  
bool resolve_cached (naming::gid_type const &id, naming::address &addr, error_code  
&ec = throws)  
  
bool resolve_cached (naming::id_type const &id, naming::address &addr, error_code  
&ec = throws)  
  
bool resolve_local (naming::gid_type const *gids, naming::address *addrs, std::size_t  
size, boost::dynamic_bitset<> &locals, error_code &ec = throws)  
  
bool resolve_full_local (naming::gid_type const *gids, naming::address *addrs,  
std::size_t size, boost::dynamic_bitset<> &locals, error_code &ec = throws)  
  
bool resolve_cached (naming::gid_type const *gids, naming::address *addrs, std::size_t  
size, boost::dynamic_bitset<> &locals, error_code &ec = throws)  
  
hpx::future<std::int64_t> incref_async (naming::gid_type const &gid, std::int64_t credits  
= 1, naming::id_type const &keep_alive = naming::invalid_id)
```

Increment the global reference count for the given id.

Return Whether the operation was successful.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *id*: [in] The global address (id) for which the global reference count has to be incremented.
- *credits*: [in] The number of reference counts to add for the given id.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
std::int64_t incref (naming::gid_type const &gid, std::int64_t credits = 1, error_code &ec =  
throws)
```

```
void decref (naming::gid_type const &id, std::int64_t credits = 1, error_code &ec = throws)
```

Decrement the global reference count for the given id.

Return The global reference count after the decrement.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *id*: [in] The global address (id) for which the global reference count has to be decremented.
- *t*: [out] If this was the last outstanding global reference for the given gid (the return value of this function is zero), t will be set to the component type of the corresponding element. Otherwise t will not be modified.
- *credits*: [in] The number of reference counts to add for the given id.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
hpx::future<iterate_names_return_type> iterate_ids (std::string const &pattern)
```

Invoke the supplied *hpx::function* for every registered global name.

This function iterates over all registered global ids and returns every found entry matching the given name pattern. Any error results in an exception thrown (or reported) from this function.

Parameters

- pattern: [in] pattern (possibly using wildcards) to match all existing entries against

```
bool register_name(std::string const &name, naming::gid_type const &id, error_code &ec = throws)
```

Register a global name with a global address (id)

This function registers an association between a global name (string) and a global address (id) usable with one of the functions above (bind, unbind, and resolve).

Return The function returns *true* if the global name was registered. It returns false if the global name is not registered.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- name: [in] The global name (string) to be associated with the global address.
- id: [in] The global address (id) to be associated with the global address.
- ec: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
hpx::future<bool> register_name_async(std::string const &name, naming::id_type const &id)
```

```
bool register_name(std::string const &name, naming::id_type const &id, error_code &ec = throws)
```

```
hpx::future<naming::id_type> unregister_name_async(std::string const &name)
```

Unregister a global name (release any existing association)

This function releases any existing association of the given global name with a global address (id).

Return The function returns *true* if an association of this global name has been released, and it returns *false*, if no association existed. Any error results in an exception thrown from this function.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- name: [in] The global name (string) for which any association with a global address (id) has to be released.
- ec: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
naming::id_type unregister_name(std::string const &name, error_code &ec = throws)
```

```
hpx::future<naming::id_type> resolve_name_async(std::string const &name)
```

Query for the global address associated with a given global name.

This function returns the global address associated with the given global name.

This function returns true if it returned global address (id), which is currently associated with the given global name, and it returns false, if currently there is no association for this global name. Any error results in an exception thrown from this function.

Return [out] The id currently associated with the given global name (valid only if the return value is true).

Parameters

- name: [in] The global name (string) for which the currently associated global address has to be retrieved.
- ec: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

`naming::id_type resolve_name (std::string const &name, error_code &ec = throws)`

`future<hpx::id_type> on_symbol_namespace_event (std::string const &name, bool call_for_past_events = false)`

Install a listener for a given symbol namespace event.

This function installs a listener for a given symbol namespace event. It returns a future which becomes ready as a result of the listener being triggered.

Return A future instance encapsulating the global id which is causing the registered listener to be triggered.

Note The only event type which is currently supported is `symbol_ns_bind`, i.e. the listener is triggered whenever a global id is registered with the given name.

Parameters

- name: [in] The global name (string) for which the given event should be triggered.
- evt: [in] The event for which a listener should be installed.
- call_for_past_events: [in, optional] Trigger the listener even if the given event has already happened in the past. The default for this parameter is `false`.

`void update_cache_entry (naming::gid_type const &gid, gva const &gva, error_code &ec = throws)`

Warning This function is for internal use only. It is dangerous and may break your code if you use it.

`void update_cache_entry (naming::gid_type const &gid, naming::address const &addr, std::uint64_t count = 0, std::uint64_t offset = 0, error_code &ec = throws)`

Warning This function is for internal use only. It is dangerous and may break your code if you use it.

`bool get_cache_entry (naming::gid_type const &gid, gva &gva, naming::gid_type &id_base, error_code &ec = throws)`

Warning This function is for internal use only. It is dangerous and may break your code if you use it.

`void remove_cache_entry (naming::gid_type const &id, error_code &ec = throws)`

Warning This function is for internal use only. It is dangerous and may break your code if you use it.

`void clear_cache (error_code &ec = throws)`

Warning This function is for internal use only. It is dangerous and may break your code if you use it.

`void start_shutdown (error_code &ec = throws)`

```

hpx::future<std::pair<naming::id_type, naming::address>> begin_migration (naming::id_type
const
&id)
    start/stop migration of an object

Return Current locality and address of the object to migrate

bool end_migration (naming::id_type const &id)

std::pair<bool, components::pinned_ptr> was_object_migrated (naming::gid_type
const &gid,
util::unique_function_nonser<components::pin
> &&fMaintain list of migrated objects.

hpx::future<void> mark_as_migrated (naming::gid_type const &gid,
util::unique_function_nonser<std::pair<bool,
hpx::future<void>>)
> &&fbool expect_to_be_marked_as_migratingMark the given object as being migrated (if the
object is unpinned). Delay migration until the object is unpinned otherwise.

void unmark_as_migrated (naming::gid_type const &gid)
    Remove the given object from the table of migrated objects.

void pre_cache_endpoints (std::vector<parcelset::endpoints_type> const&)

```

Public Members

```

mutex_type gva_cache_mtx_
std::shared_ptr<gva_cache_type> gva_cache_
mutex_type migrated_objects_mtx_
migrated_objects_table_type migrated_objects_table_
mutex_type console_cache_mtx_
std::uint32_t console_cache_
const std::size_t max_refcnt_requests_
mutex_type refcnt_requests_mtx_
std::size_t refcnt_requests_count_
bool enable_refcnt_caching_
std::shared_ptr<refcnt_requests_type> refcnt_requests_
const service_mode service_type
const runtime_mode runtime_type
const bool caching_
const bool range_caching_
const threads::thread_priority action_priority_
std::uint64_t rts_lva_
std::unique_ptr<component_namespace> component_ns_
std::unique_ptr<locality_namespace> locality_ns_

```

```
symbol_namespace symbol_ns_
primary_namespace primary_ns_
std::atomic<hpx::state> state_
naming::gid_type locality_
mutex_type resolved_localities_mtx_
resolved_localities_type resolved_localities_
```

Protected Functions

```
void launch_bootstrap (parcelset::endpoints_type const &endpoints,
                      util::runtime_configuration &rtecfg)

naming::address resolve_full_postproc (naming::gid_type const &id, future<primary_namespace::resolved_type> f)

bool bind_postproc (naming::gid_type const &id, gva const &g, future<bool> f)

bool was_object_migrated_locked (naming::gid_type const &id)
    Maintain list of migrated objects.
```

Private Functions

```
void send_refcnt_requests (std::unique_lock<mutex_type> &l, error_code &ec = throws)
    Assumes that refcnt_requests_mtx_ is locked.

void send_refcnt_requests_non_blocking (std::unique_lock<mutex_type> &l, error_code &ec)
    Assumes that refcnt_requests_mtx_ is locked.

std::vector<hpx::future<std::vector<std::int64_t>>> send_refcnt_requests_async (std::unique_lock<mutex_type> &l)
    Assumes that refcnt_requests_mtx_ is locked.

void send_refcnt_requests_sync (std::unique_lock<mutex_type> &l, error_code &ec)
    Assumes that refcnt_requests_mtx_ is locked.
```

namespace hpx

namespace naming

TypeDefs

```
using resolver_client = agas::addressing_service
```

Functions

```
agas::addressing_service &get_agas_client()
agas::addressing_service *get_agas_client_ptr()

namespace hpx

namespace agas
```

Functions

```
bool router_is(state st)
```

agas_base

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace agas
```

Typedefs

```
using iterate_types_function_type = hpx::util::function<void (std::string
                                                       const&,           components::component_type)>
```

Variables

```
constexpr char const *const service_name = "/0/agas/"
constexpr const std::uint64_t bootstrap_prefix = 0ULL
constexpr const std::uint64_t primary_ns_msb = 0x100000001ULL
constexpr const std::uint64_t primary_ns_lsb = 0x000000001ULL
constexpr const std::uint64_t component_ns_msb = 0x100000001ULL
constexpr const std::uint64_t component_ns_lsb = 0x000000002ULL
constexpr const std::uint64_t symbol_ns_msb = 0x100000001ULL
constexpr const std::uint64_t symbol_ns_lsb = 0x000000003ULL
constexpr const std::uint64_t locality_ns_msb = 0x100000001ULL
constexpr const std::uint64_t locality_ns_lsb = 0x000000004ULL
```

namespace server

AGAS's primary namespace maps 128-bit global identifiers (GIDs) to resolved addresses.

The following is the canonical description of the partitioning of AGAS's primary namespace.

	-----MSB----- -----LSB-----
	BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
	prefix RC ----identifier----
MSB	- Most significant bits (bit 64 to bit 127)
LSB	- Least significant bits (bit 0 to bit 63)
prefix	- Highest 32 bits (bit 96 to bit 127) of the MSB. Each locality is assigned a prefix. This creates a 96-bit address space for each locality.
RC	- Bit 88 to bit 92 of the MSB. This is the log ₂ of the number of reference counting credits on the GID. Bit 93 is used by the locking scheme for gid_types. Bit 94 is a flag which is set if the credit value is valid. Bit 95 is a flag that is set if a GID's credit count is ever split (e.g. if the GID is ever passed to another locality). - Bit 87 marks the gid such that it will not be stored in any of the AGAS caches. This is used mainly for ids which represent 'one-shot' objects (like promises).
identifier	- Bit 64 to bit 86 of the MSB, and the entire LSB. The content of these bits depends on the component type of the underlying object. For all user-defined components, these bits contain a unique 88-bit number which is assigned sequentially for each locality. For \a hpx#components#component_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

Note The layout of the address space is implementation defined, and subject to change. Never write application code that relies on the internal layout of GIDs. AGAS only guarantees that all assigned GIDs will be unique.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

00000000xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	Historically unused address space reserved for future use.
xxxxxxxxxxxx000xxxxxxxxxxxxxxxxxxxx	Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxxxxxx	Prefix of the bootstrap AGAS locality.
0000000100000000100000000000000001	Address of the primary_namespace component on the bootstrap AGAS locality.
000000010000000010000000000000000002	Address of the component_namespace component on the bootstrap AGAS locality.
000000010000000010000000000000000003	Address of the symbol_namespace component on the bootstrap AGAS locality.
000000010000000010000000000000000004	Address of the locality_namespace component on the bootstrap AGAS locality.

namespace components

Typedefs

```
using component_type = std::int32_t
namespace hpx
```

```
namespace agas
```

```
struct component_namespace
```

Public Functions

```
virtual ~component_namespace()
virtual naming::address::address_type ptr() const = 0
virtual naming::address addr() const = 0
virtual naming::id_type gid() const = 0
virtual components::component_type bind_prefix(std::string const &key,
                                                std::uint32_t prefix) = 0
virtual components::component_type bind_name(std::string const &name) = 0
virtual std::vector<std::uint32_t> resolve_id(components::component_type key) = 0
virtual bool unbind(std::string const &key) = 0
virtual void iterate_types(iterate_types_function_type const &f) = 0
virtual std::string get_component_type_name(components::component_type type) =
    0
virtual hpx::future<std::uint32_t> get_num_localities(components::component_type
                                                       type) = 0
virtual void register_server_instance(std::uint32_t)
virtual void unregister_server_instance(error_code&
                                         server::component_namespace *get_service()
                                         )
namespace hpx

namespace agas
```

Functions

```
std::ostream &operator<< (std::ostream &os, gva const &addr)  
  
struct gva
```

Public Types

```
using component_type = std::int32_t  
using lva_type = void*
```

Public Functions

```
constexpr gva ()  
  
constexpr gva (naming::gid_type const &p, component_type t = components::component_invalid, std::uint64_t c = 1, lva_type a = nullptr, std::uint64_t o = 0)  
  
gva (naming::gid_type const &p, component_type t, std::uint64_t c, std::uint64_t l, std::uint64_t o = 0)  
  
constexpr gva (lva_type a)  
  
gva &operator=(lva_type a)  
  
constexpr bool operator==(gva const &rhs) const  
  
constexpr bool operator!=(gva const &rhs) const  
  
void lva (lva_type a)  
  
constexpr lva_type lva () const  
  
lva_type lva (naming::gid_type const &gid, naming::gid_type const &gidbase) const  
gva resolve (naming::gid_type const &gid, naming::gid_type const &gidbase) const
```

Public Members

```
naming::gid_type prefix  
component_type type = components::component_invalid  
std::uint64_t count = 0  
std::uint64_t offset = 0
```

Private Functions

```
template<typename Archive>
void save(Archive &ar, const unsigned int) const

template<typename Archive>
void load(Archive &ar, const unsigned int version)
```

Private Members

```
lva_type lva_ = nullptr
```

Friends

```
friend hpx::agas::hpx::serialization::access

namespace hpx

namespace agas

struct locality_namespace
```

Public Functions

```
virtual ~locality_namespace()

virtual naming::address::address_type ptr() const = 0

virtual naming::address addr() const = 0

virtual naming::id_type gid() const = 0

virtual std::uint32_t allocate(parcelset::endpoints_type const &endpoints,
                           std::uint64_t count, std::uint32_t num_threads, naming::gid_type const &suggested_prefix) = 0

virtual void free(naming::gid_type const &locality) = 0

virtual std::vector<std::uint32_t> localities() = 0

virtual parcelset::endpoints_type resolve_locality(naming::gid_type const &locality) = 0

virtual std::uint32_t get_num_localities() = 0

virtual hpx::future<std::uint32_t> get_num_localities_async() = 0

virtual std::vector<std::uint32_t> get_num_threads() = 0

virtual hpx::future<std::vector<std::uint32_t>> get_num_threads_async() = 0

virtual std::uint32_t get_num_overall_threads() = 0

virtual hpx::future<std::uint32_t> get_num_overall_threads_async() = 0
```

```
    virtual void register_server_instance (std::uint32_t)
    virtual void unregister_server_instance (error_code&)
    virtual server::locality_namespace *get_service ()

namespace hpx
```

```
namespace agas
```

```
struct primary_namespace
```

Public Types

```
typedef hpx::tuple<naming::gid_type, gva, naming::gid_type> resolved_type
```

Public Functions

```
primary_namespace ()
```

```
~primary_namespace ()
```

```
naming::address::address_type ptr () const
```

```
naming::address addr () const
```

```
naming::id_type gid () const
```

```
hpx::future<std::pair<naming::id_type, naming::address>> begin_migration (naming::gid_type
                                                                           const
                                                                           &id)
```

```
bool end_migration (naming::gid_type const &id)
```

```
bool bind_gid (gva const &g, naming::gid_type const &id, naming::gid_type const &lo-
                cality)
```

```
future<bool> bind_gid_async (gva g, naming::gid_type id, naming::gid_type locality)
```

```
resolved_type resolve_gid (naming::gid_type const &id)
```

```
future<resolved_type> resolve_full (naming::gid_type id)
```

```
future<id_type> colocate (naming::gid_type id)
```

```
naming::address unbind_gid (std::uint64_t count, naming::gid_type const &id)
```

```
future<naming::address> unbind_gid_async (std::uint64_t count, naming::gid_type
                                            const &id)
```

```
future<std::int64_t> increment_credit (std::int64_t credits, naming::gid_type lower, nam-
                                         ing::gid_type upper)
```

```
std::pair<naming::gid_type, naming::gid_type> allocate (std::uint64_t count)
```

```
void set_local_locality (naming::gid_type const &g)
```

```
void register_server_instance(std::uint32_t locality_id)
void unregister_server_instance(error_code &ec)
server::primary_namespace &get_service()
```

Public Static Functions

```
static naming::gid_type get_service_instance(std::uint32_t service_locality_id)
static naming::gid_type get_service_instance(naming::gid_type const &dest, error_code &ec = throws)
static naming::gid_type get_service_instance(naming::id_type const &dest)
static bool is_service_instance(naming::gid_type const &gid)
static bool is_service_instance(naming::id_type const &id)
```

Private Members

```
std::unique_ptr<server::primary_namespace> server_
namespace hpx

namespace agas

struct symbol_namespace
```

Public Types

```
using server_type = server::symbol_namespace
using iterate_names_return_type = std::map<std::string, naming::id_type>
```

Public Functions

```
symbol_namespace()
~symbol_namespace()

naming::address_type ptr() const
naming::address addr() const
naming::id_type gid() const

hpx::future<bool> bind_async(std::string key, naming::gid_type gid)
bool bind(std::string key, naming::gid_type gid)

hpx::future<naming::id_type> resolve_async(std::string key) const
naming::id_type resolve(std::string key) const
```

```
hpx::future<naming::id_type> unbind_async (std::string key)
naming::id_type unbind (std::string key)

hpx::future<bool> on_event (std::string const &name, bool call_for_past_events,
                           hpx::id_type lco)

hpx::future<iterate_names_return_type> iterate_async (std::string const &pattern)
                                                       const
iterate_names_return_type iterate (std::string const &pattern) const

void register_server_instance (std::uint32_t locality_id)

void unregister_server_instance (error_code &ec)

server::symbol_namespace &get_service ()
```

Public Static Functions

```
static naming::gid_type get_service_instance (std::uint32_t service_locality_id)

static naming::gid_type get_service_instance (naming::gid_type const &dest, error_code &ec = throws)

static naming::gid_type get_service_instance (naming::id_type const &dest)

static bool is_service_instance (naming::gid_type const &gid)

static bool is_service_instance (naming::id_type const &id)

static naming::id_type symbol_namespace_locality (std::string const &key)
```

Private Members

```
std::unique_ptr<server_type> server_

namespace hpx

namespace agas
```

Functions

```
naming::gid_type bootstrap_component_namespace_gid ()
```

```
naming::id_type bootstrap_component_namespace_id ()
```

```
namespace server
```

AGAS's primary namespace maps 128-bit global identifiers (GIDs) to resolved addresses.

The following is the canonical description of the partitioning of AGAS's primary namespace.

```

|-----MSB-----| |-----LSB-----|
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
|prefix||RC||----identifier----|

MSB      - Most significant bits (bit 64 to bit 127)
LSB      - Least significant bits (bit 0 to bit 63)
prefix   - Highest 32 bits (bit 96 to bit 127) of the MSB. Each
           locality is assigned a prefix. This creates a 96-bit
           address space for each locality.
RC       - Bit 88 to bit 92 of the MSB. This is the log2 of the number
           of reference counting credits on the GID.
           Bit 93 is used by the locking scheme for gid_types.
           Bit 94 is a flag which is set if the credit value is valid.
           Bit 95 is a flag that is set if a GID's credit count is
           ever split (e.g. if the GID is ever passed to another
           locality).
           - Bit 87 marks the gid such that it will not be stored in
             any of the AGAS caches. This is used mainly for ids
             which represent 'one-shot' objects (like promises).
identifier - Bit 64 to bit 86 of the MSB, and the entire LSB. The
           content of these bits depends on the component type of
           the underlying object. For all user-defined components,
           these bits contain a unique 88-bit number which is
           assigned sequentially for each locality. For
           \a hpx#components#component_runtime_support the high 24
           bits are zeroed and the low 64 bits hold the LVA of the
           component.

```

Note The layout of the address space is implementation defined, and subject to change. Never write application code that relies on the internal layout of GIDs. AGAS only guarantees that all assigned GIDs will be unique.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

```

00000000xxxxxxxxxxxxxxxxxxxxxxxxxx
    Historically unused address space reserved for future use.
xxxxxxxxxxxx000xxxxxxxxxxxxxxxxx
    Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxx
    Prefix of the bootstrap AGAS locality.
00000001000000001000000000000001
    Address of the primary_namespace component on the bootstrap AGAS
    locality.
000000010000000010000000000000002
    Address of the component_namespace component on the bootstrap AGAS
    locality.
000000010000000010000000000000003
    Address of the symbol_namespace component on the bootstrap AGAS
    locality.
000000010000000010000000000000004
    Address of the locality_namespace component on the bootstrap AGAS
    locality.

```

Variables

```
constexpr char const *const component_namespace_service_name = "component/"

struct component_namespace : public components::fixed_component_base<component_namespace>
```

Public Types

```
using mutex_type = lcos::local::spinlock
using base_type = components::fixed_component_base<component_namespace>
using component_id_type = components::component_type
using prefixes_type = std::set<std::uint32_t>
using component_id_table_type = std::unordered_map<std::string, component_id_type>
using factory_table_type = std::map<component_id_type, prefixes_type>
```

Public Functions

```
component_namespace()
void finalize()
void register_server_instance(char const *servicename, error_code &ec =
    throws)
void unregister_server_instance(error_code &ec = throws)
components::component_type bind_prefix(std::string const &key, std::uint32_t prefix)
components::component_type bind_name(std::string const &name)
std::vector<std::uint32_t> resolve_id(components::component_type key)
bool unbind(std::string const &key)
void iterate_types(iterate_types_function_type const &f)
std::string get_component_type_name(components::component_type type)
std::uint32_t get_num_localities(components::component_type type)
```

Public Members

counter_data counter_data_

Public Static Functions

```
static void register_counter_types(error_code &ec = throws)
static void register_global_counter_types(error_code &ec = throws)
```

Private Members

```
mutex_type mutex_
component_id_table_type component_ids_
factory_table_type factories_
component_id_type type_counter
std::string instance_name_
struct counter_data
```

Public Types

```
typedef lcos::local::spinlock mutex_type
```

Public Functions

```
HPX_NON_COPYABLE(counter_data)

counter_data()

std::int64_t get_bind_prefix_count(bool)
std::int64_t get_bind_name_count(bool)
std::int64_t get_resolve_id_count(bool)
std::int64_t get_unbind_name_count(bool)
std::int64_t get_iterate_types_count(bool)
std::int64_t get_component_type_name_count(bool)
std::int64_t get_num_localities_count(bool)
std::int64_t get_overall_count(bool)
std::int64_t get_bind_prefix_time(bool)
std::int64_t get_bind_name_time(bool)
std::int64_t get_resolve_id_time(bool)
std::int64_t get_unbind_name_time(bool)
std::int64_t get_iterate_types_time(bool)
std::int64_t get_component_type_name_time(bool)
```

```
    std::int64_t get_num_localities_time(bool)
    std::int64_t get_overall_time(bool)

    void increment_bind_prefix_count()
    void increment_bind_name_count()
    void increment_resolve_id_count()
    void increment_unbind_name_count()
    void increment_iterate_types_count()
    void increment_get_component_type_name_count()
    void increment_num_localities_count()
    void enable_all()
```

Public Members

```
api_counter_data bind_prefix_
api_counter_data bind_name_
api_counter_data resolve_id_
api_counter_data unbind_name_
api_counter_data iterate_types_
api_counter_data get_component_type_name_
api_counter_data num_localities_

struct api_counter_data
```

Public Functions

```
api_counter_data()
```

Public Members

```
std::atomic<std::int64_t> count_
std::atomic<std::int64_t> time_
bool enabled_

namespace hpx

namespace agas
```

Functions

`naming::gid_type bootstrap_locality_namespace_gid()`

`naming::id_type bootstrap_locality_namespace_id()`

namespace server

AGAS's primary namespace maps 128-bit global identifiers (GIDs) to resolved addresses.

The following is the canonical description of the partitioning of AGAS's primary namespace.

-----MSB----- -----LSB-----
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
prefix RC ---identifier---
MSB - Most significant bits (bit 64 to bit 127)
LSB - Least significant bits (bit 0 to bit 63)
prefix - Highest 32 bits (bit 96 to bit 127) of the MSB. Each locality is assigned a prefix. This creates a 96-bit address space for each locality.
RC - Bit 88 to bit 92 of the MSB. This is the log2 of the number of reference counting credits on the GID. Bit 93 is used by the locking scheme for gid_types. Bit 94 is a flag which is set if the credit value is valid. Bit 95 is a flag that is set if a GID's credit count is ever split (e.g. if the GID is ever passed to another locality). - Bit 87 marks the gid such that it will not be stored in any of the AGAS caches. This is used mainly for ids which represent 'one-shot' objects (like promises).
identifier - Bit 64 to bit 86 of the MSB, and the entire LSB. The content of these bits depends on the component type of the underlying object. For all user-defined components, these bits contain a unique 88-bit number which is assigned sequentially for each locality. For \a hpx#components#component_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

Note The layout of the address space is implementation defined, and subject to change. Never write application code that relies on the internal layout of GIDs. AGAS only guarantees that all assigned GIDs will be unique.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

00000000xxxxxxxxxxxxxxxxxxxxxx
Historically unused address space reserved for future use.
xxxxxxxxxxxx000xxxxxxxxxxxxxx
Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxx
Prefix of the bootstrap AGAS locality.
00000001000000001000000000000001
Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000001000000000000002

(continues on next page)

(continued from previous page)

```
Address of the component_namespace component on the bootstrap AGAS
locality.  
000000010000000100000000000000000003  
Address of the symbol_namespace component on the bootstrap AGAS
locality.  
000000010000000100000000000000000004  
Address of the locality_namespace component on the bootstrap AGAS
locality.
```

Variables

```
constexpr char const *const locality_namespace_service_name = "locality"  
  
struct locality_namespace : public components::fixed_component_base<locality_namespace>
```

Public Types

```
using mutex_type = lcos::local::spinlock
using base_type = components::fixed_component_base<locality_namespace>
using component_type = std::int32_t
using partition_type = hpx::tuple<parcelset::endpoints_type, std::uint32_t>
using partition_table_type = std::map<std::uint32_t, partition_type>
```

Public Functions

```
locality_namespace (primary_namespace *primary)

void finalize()

void register_server_instance (char const *servicename, error_code &ec =
                               throws)

void unregister_server_instance (error_code &ec = throws)

std::uint32_t allocate (parcelset::endpoints_type const &endpoints, std::uint64_t count,
                       std::uint32_t num_threads, naming::gid_type suggested_prefix)

parcelset::endpoints_type resolve_locality (naming::gid_type const &locality)

void free (naming::gid_type const &locality)

std::vector<std::uint32_t> localities ()

std::uint32_t get_num_localities ()

std::vector<std::uint32_t> get_num_threads ()

std::uint32_t get_num_overall_threads ()
```

Public Members

counter_data **counter_data_**

Private Members

mutex_type **mutex_**
std::string **instance_name_**
partition_table_type **partitions_**
std::uint32_t **prefix_counter_**
primary_namespace ***primary_**
struct counter_data

Public Types

typedef lcos::local::spinlock mutex_type

Public Functions

HPX_NON_COPYABLE (*counter_data*)
counter_data()
std::int64_t **get_allocate_count** (bool)
std::int64_t **get_resolve_locality_count** (bool)
std::int64_t **get_free_count** (bool)
std::int64_t **get_localities_count** (bool)
std::int64_t **get_num_localities_count** (bool)
std::int64_t **get_num_threads_count** (bool)
std::int64_t **get_resolved_localities_count** (bool)
std::int64_t **get_overall_count** (bool)
std::int64_t **get_allocate_time** (bool)
std::int64_t **get_resolve_locality_time** (bool)
std::int64_t **get_free_time** (bool)
std::int64_t **get_localities_time** (bool)
std::int64_t **get_num_localities_time** (bool)
std::int64_t **get_num_threads_time** (bool)
std::int64_t **get_resolved_localities_time** (bool)

```
    std::int64_t get_overall_time(bool)
    void increment_allocate_count()
    void increment_resolve_locality_count()
    void increment_free_count()
    void increment_localities_count()
    void increment_num_localities_count()
    void increment_num_threads_count()
    void enable_all()
```

Public Members

```
api_counter_data allocate_
api_counter_data resolve_locality_
api_counter_data free_
api_counter_data localities_
api_counter_data num_localities_
api_counter_data num_threads_
struct api_counter_data
```

Public Functions

```
api_counter_data()
```

Public Members

```
std::atomic<std::int64_t> count_
std::atomic<std::int64_t> time_
bool enabled_
```

Variables

```
HPX_ACTIONUSES_MEDIUMSTACK ( hpx::agas::server::primary_namespace::allocate_action) HPX_
namespace hpx

namespace agas
```

Functions

`naming::gid_type bootstrap_primary_namespace_gid()`

`naming::id_type bootstrap_primary_namespace_id()`

namespace server

AGAS's primary namespace maps 128-bit global identifiers (GIDs) to resolved addresses.

The following is the canonical description of the partitioning of AGAS's primary namespace.

-----MSB----- -----LSB-----
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
prefix RC ---identifier---
MSB - Most significant bits (bit 64 to bit 127)
LSB - Least significant bits (bit 0 to bit 63)
prefix - Highest 32 bits (bit 96 to bit 127) of the MSB. Each locality is assigned a prefix. This creates a 96-bit address space for each locality.
RC - Bit 88 to bit 92 of the MSB. This is the log2 of the number of reference counting credits on the GID. Bit 93 is used by the locking scheme for gid_types. Bit 94 is a flag which is set if the credit value is valid. Bit 95 is a flag that is set if a GID's credit count is ever split (e.g. if the GID is ever passed to another locality). - Bit 87 marks the gid such that it will not be stored in any of the AGAS caches. This is used mainly for ids which represent 'one-shot' objects (like promises).
identifier - Bit 64 to bit 86 of the MSB, and the entire LSB. The content of these bits depends on the component type of the underlying object. For all user-defined components, these bits contain a unique 88-bit number which is assigned sequentially for each locality. For \a hpx#components#component_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

Note The layout of the address space is implementation defined, and subject to change. Never write application code that relies on the internal layout of GIDs. AGAS only guarantees that all assigned GIDs will be unique.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

00000000xxxxxxxxxxxxxxxxxxxxxx
Historically unused address space reserved for future use.
xxxxxxxxxxxx000xxxxxxxxxxxxxx
Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxx
Prefix of the bootstrap AGAS locality.
00000001000000001000000000000001
Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000001000000000000002

(continues on next page)

(continued from previous page)

```
Address of the component_namespace component on the bootstrap AGAS
locality.  
000000010000000100000000000000000003  
Address of the symbol_namespace component on the bootstrap AGAS
locality.  
000000010000000100000000000000000004  
Address of the locality_namespace component on the bootstrap AGAS
locality.
```

Variables

```
constexpr char const *const primary_namespace_service_name = "primary"/"

struct primary_namespace : public components::fixed_component_base<primary_namespace>
```

Public Types

```
using mutex_type = lcos::local::spinlock
using base_type = components::fixed_component_base<primary_namespace>
using component_type = std::int32_t
using gva_table_data_type = std::pair<gva, naming::gid_type>
using gva_table_type = std::map<naming::gid_type, gva_table_data_type>
using refcnt_table_type = std::map<naming::gid_type, std::int64_t>
using resolved_type = hpx::tuple<naming::gid_type, gva, naming::gid_type>
```

Public Functions

```
primary_namespace()

void finalize()

void set_local_locality(naming::gid_type const &g)

void register_server_instance(char const *servicename, std::uint32_t locality_id
                             = naming::invalid_locality_id, error_code &ec =
                             throws)

void unregister_server_instance(error_code &ec = throws)

bool bind_gid(gva const &g, naming::gid_type id, naming::gid_type const &locality)
std::pair<naming::id_type, naming::address> begin_migration(naming::gid_type id)

bool end_migration(naming::gid_type const &id)

resolved_type resolve_gid(naming::gid_type const &id)

naming::id_type colocate(naming::gid_type const &id)

naming::address unbind_gid(std::uint64_t count, naming::gid_type id)
```

```

std::int64_t increment_credit (std::int64_t credits, naming::gid_type lower, naming::gid_type upper)

std::vector<std::int64_t> decrement_credit (std::vector<hpx::tuple<std::int64_t, naming::gid_type, naming::gid_type>> const &requests)

std::pair<naming::gid_type, naming::gid_type> allocate (std::uint64_t count)

```

Public Members

counter_data **counter_data_**

Private Types

```

using migration_table_type = std::map<naming::gid_type, hpx::tuple<bool, std::size_t, lcos::local::data>>
using free_entry_allocator_type = util::internal_allocator<free_entry>
using free_entry_list_type = std::list<free_entry, free_entry_allocator_type>

```

Private Functions

```

void wait_for_migration_locked (std::unique_lock<mutex_type> &l, naming::gid_type const &id, error_code &ec)

resolved_type resolve_gid_locked (std::unique_lock<mutex_type> &l, naming::gid_type const &gid, error_code &ec)

void increment (naming::gid_type const &lower, naming::gid_type const &upper, std::int64_t &credits, error_code &ec)

void resolve_free_list (std::unique_lock<mutex_type> &l, std::list<refcnt_table_type::iterator> const &free_list, free_entry_list_type &free_entry_list, naming::gid_type const &lower, naming::gid_type const &upper, error_code &ec)

void decrement_sweep (free_entry_list_type &free_list, naming::gid_type const &lower, naming::gid_type const &upper, std::int64_t credits, error_code &ec)

void free_components_sync (free_entry_list_type &free_list, naming::gid_type const &lower, naming::gid_type const &upper, error_code &ec)

```

Private Members

mutex_type **mutex_**
gva_table_type **gvabs_**
refcnt_table_type **refcnts_**
std::string **instance_name_**
naming::gid_type **next_id_**

```
naming::gid_type locality_
migration_table_type migrating_objects_

struct counter_data
```

Public Functions

```
HPX_NON_COPYABLE(counter_data)

counter_data()

std::int64_t get_bind_gid_count(bool)
std::int64_t get_resolve_gid_count(bool)
std::int64_t get_unbind_gid_count(bool)
std::int64_t get_increment_credit_count(bool)
std::int64_t get_decrement_credit_count(bool)
std::int64_t get_allocate_count(bool)
std::int64_t get_begin_migration_count(bool)
std::int64_t get_end_migration_count(bool)
std::int64_t get_overall_count(bool)
std::int64_t get_bind_gid_time(bool)
std::int64_t get_resolve_gid_time(bool)
std::int64_t get_unbind_gid_time(bool)
std::int64_t get_increment_credit_time(bool)
std::int64_t get_decrement_credit_time(bool)
std::int64_t get_allocate_time(bool)
std::int64_t get_begin_migration_time(bool)
std::int64_t get_end_migration_time(bool)
std::int64_t get_overall_time(bool)

void increment_bind_gid_count()
void increment_resolve_gid_count()
void increment_unbind_gid_count()
void increment_increment_credit_count()
void increment_decrement_credit_count()
void increment_allocate_count()
void increment_begin_migration_count()
```

```
void increment_end_migration_count()
void enable_all()
```

Public Members

```
api_counter_data bind_gid_
api_counter_data resolve_gid_
api_counter_data unbind_gid_
api_counter_data increment_credit_
api_counter_data decrement_credit_
api_counter_data allocate_
api_counter_data begin_migration_
api_counter_data end_migration_
struct api_counter_data
```

Public Functions

```
api_counter_data()
```

Public Members

```
std::atomic<std::int64_t> count_
std::atomic<std::int64_t> time_
bool enabled_
struct free_entry
```

Public Functions

```
free_entry(agas::gva gva, naming::gid_type const &gid, naming::gid_type const
&loc)
```

Public Members

```
agas::gva gva_
naming::gid_type gid_
naming::gid_type locality_
namespace hpx
namespace agas
```

Functions

`naming::gid_type bootstrap_symbol_namespace_gid()`

`naming::id_type bootstrap_symbol_namespace_id()`

namespace server

AGAS's primary namespace maps 128-bit global identifiers (GIDs) to resolved addresses.

The following is the canonical description of the partitioning of AGAS's primary namespace.

-----MSB----- -----LSB-----
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
prefix RC ---identifier---
MSB - Most significant bits (bit 64 to bit 127)
LSB - Least significant bits (bit 0 to bit 63)
prefix - Highest 32 bits (bit 96 to bit 127) of the MSB. Each locality is assigned a prefix. This creates a 96-bit address space for each locality.
RC - Bit 88 to bit 92 of the MSB. This is the log2 of the number of reference counting credits on the GID. Bit 93 is used by the locking scheme for gid_types. Bit 94 is a flag which is set if the credit value is valid. Bit 95 is a flag that is set if a GID's credit count is ever split (e.g. if the GID is ever passed to another locality). - Bit 87 marks the gid such that it will not be stored in any of the AGAS caches. This is used mainly for ids which represent 'one-shot' objects (like promises).
identifier - Bit 64 to bit 86 of the MSB, and the entire LSB. The content of these bits depends on the component type of the underlying object. For all user-defined components, these bits contain a unique 88-bit number which is assigned sequentially for each locality. For \a hpx#components#component_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

Note The layout of the address space is implementation defined, and subject to change. Never write application code that relies on the internal layout of GIDs. AGAS only guarantees that all assigned GIDs will be unique.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

00000000xxxxxxxxxxxxxxxxxxxxxx
Historically unused address space reserved for future use.
xxxxxxxxxxxx000xxxxxxxxxxxxxx
Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxx
Prefix of the bootstrap AGAS locality.
00000001000000010000000000000001
Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000010000000000000002

(continues on next page)

(continued from previous page)

```
Address of the component_namespace component on the bootstrap AGAS  
locality.  
00000001000000010000000000000000  
Address of the symbol_namespace component on the bootstrap AGAS  
locality.  
00000001000000010000000000000004  
Address of the locality_namespace component on the bootstrap AGAS  
locality.
```

Variables

```
constexpr char const *const symbol_namespace_service_name = "symbol/"

struct symbol_namespace : public components::fixed_component_base<symbol_namespace>
```

Public Types

```
using mutex_type = lcos::local::spinlock
using base_type = components::fixed_component_base<symbol_namespace>
using iterate_names_return_type = std::map<std::string, naming::gid_type>
using gid_table_type = std::map<std::string, std::shared_ptr<naming::gid_type>>
using on_event_data_map_type = std::multimap<std::string, hpx::id_type>
```

Public Functions

```
symbol_namespace()

void finalize()

void register_server_instance(char const *servicename, std::uint32_t locality_id
    = naming::invalid_locality_id, error_code &ec =
throws)

void unregister_server_instance(error_code &ec = throws)

bool bind(std::string key, naming::gid_type gid)

naming::gid_type resolve(std::string const &key)

naming::gid_type unbind(std::string const &key)

iterate_names_return_type iterate(std::string const &pattern)

bool on_event(std::string const &name, bool call_for_past_events, hpx::id_type lco)
```

Public Members

counter_data **counter_data_**

Public Static Functions

static void register_counter_types (*error_code &ec = throws*)

static void register_global_counter_types (*error_code &ec = throws*)

Private Members

mutex_type **mutex_**

gid_table_type **gids_**

std::string **instance_name_**

on_event_data_map_type **on_event_data_**

struct counter_data

Public Types

typedef lcos::local::spinlock mutex_type

Public Functions

HPX_NON_COPYABLE (*counter_data*)

counter_data()

std::int64_t **get_bind_count** (bool)

std::int64_t **get_resolve_count** (bool)

std::int64_t **get_unbind_count** (bool)

std::int64_t **get_iterate_names_count** (bool)

std::int64_t **get_on_event_count** (bool)

std::int64_t **get_overall_count** (bool)

std::int64_t **get_bind_time** (bool)

std::int64_t **get_resolve_time** (bool)

std::int64_t **get_unbind_time** (bool)

std::int64_t **get_iterate_names_time** (bool)

std::int64_t **get_on_event_time** (bool)

std::int64_t **get_overall_time** (bool)

```
void increment_bind_count()  
void increment_resolve_count()  
void increment_unbind_count()  
void increment_iterate_names_count()  
void increment_on_event_count()  
void enable_all()
```

Public Members

```
api_counter_data bind_  
api_counter_data resolve_  
api_counter_data unbind_  
api_counter_data iterate_names_  
api_counter_data on_event_  
struct api_counter_data
```

Public Functions

```
api_counter_data()
```

Public Members

```
std::atomic<std::int64_t> count_  
std::atomic<std::int64_t> time_  
bool enabled_
```

async_colocated

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_REGISTER_ASYNC_COLOCATED_DECLARATION(Action, Name)  
HPX_REGISTER_ASYNC_COLOCATED(Action, Name)  
namespace hpx
```

Functions

naming::id_type get_colocation_id(launch::sync_policy, naming::id_type const &id, error_code &ec = throws)

Return the id of the locality where the object referenced by the given id is currently located on.

The function `hpx::get_colocation_id()` returns the id of the locality where the given object is currently located.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

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.

hpx::future<naming::id_type> get_colocation_id(naming::id_type const &id)

Asynchronously return the id of the locality where the object referenced by the given id is currently located on.

See `hpx::get_colocation_id(launch::sync_policy)`

Parameters

- `id`: [in] The id of the object to locate.

Defines

HPX_REGISTER_APPLY_COLOCATED_DECLARATION(*Action, Name*)

HPX_REGISTER_APPLY_COLOCATED(*action, name*)

namespace hpx

namespace util

namespace functional

Functions

template<typename Bound>
functional::detail::apply_continuation_impl<Bound, hpx::util::unused_type> apply_continuation(Bound &&bound)

template<typename Bound, typename Continuation>

```

functional::detail::apply_continuation_impl<Bound, Continuation> apply_continuation(Bound
&&bound,
Contin-
tin-
u-
a-
tion
&&c)

template<typename Bound>
functional::detail::async_continuation_impl<Bound, hpx::util::unused_type> async_continuation(Bound
&&bound)

template<typename Bound, typename Continuation>
functional::detail::async_continuation_impl<Bound, Continuation> async_continuation(Bound
&&bound,
Contin-
tin-
u-
a-
tion
&&c)

struct extract_locality

```

Public Functions

```

naming::id_type operator() (naming::id_type const &locality_id, naming::id_type
const &id) const

namespace hpx

namespace components

namespace server

```

Functions

```

void destroy_component (naming::gid_type const &gid, naming::address const
&addr)

template<typename Component>
void destroy (naming::gid_type const &gid, naming::address const &addr)

template<typename Component>
void destroy (naming::gid_type const &gid)

```

async_distributed

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx**Functions**

```
template<typename Action, typename F, typename ...Ts>
auto async (F &&f, Ts&&... ts)
```

namespace hpx**Functions**

```
template<typename Action, typename F, typename ...Ts>
auto async_cb (F &&f, Ts&&... ts)
```

```
template<typename F, typename ...Ts>
auto async_cb (F &&f, Ts&&... ts)
```

namespace hpx**Functions**

```
template<typename Action, typename Cont, typename ...Ts>
```

```
hpx::future<typename traits::promise_local_result<typename detail::result_of_async_continue<Action, Cont>::type>::type>
```

```
template<typename Component, typename Signature, typename Derived, typename Cont, typename ...Ts>
```

```
hpx::future<typename traits::promise_local_result<typename detail::result_of_async_continue<Derived, Cont>::type>::type>
```

```
template<typename Action, typename Cont, typename DistPolicy, typename ...Ts>
```

```
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, hpx::future<typename traits::promise_local_result<typename
```

```
template<typename Component, typename Signature, typename Derived, typename Cont, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, hpx::future<typename traits::promise_local_result<typename
```

```
namespace hpx
```

Functions

```
template<typename Action, typename Cont, typename Callback, typename ...Ts>
hpx::future<typename traits::promise_local_result<typename detail::result_of_async_continue<Action, Cont>::type>>
```

```
template<typename Component, typename Signature, typename Derived, typename Cont, typename Callback, typename ...Ts>
```

```
hpx::future<typename traits::promise_local_result<typename detail::result_of_async_continue<Derived, Cont>::type>>::type
```

```
template<typename Action, typename Cont, typename DistPolicy, typename Callback, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, hpx::future<typename traits::promise_local_result<typename
```

```
template<typename Component, typename Signature, typename Derived, typename Cont, typename DistPolicy, typename
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, hpx::future<typename traits::promise_local_result<typename
```

```
template<>
struct get_lva<lcos::base_lco>
```

Public Static Functions

```
static constexpr lcos::base_lco *call (naming::address_type lva)
template<>
struct get_lva<lcos::base_lco const>
```

Public Static Functions

```
static constexpr lcos::base_lco const *call (naming::address_type lva)
namespace hpx
```

```
template<>
struct get_lva<lcos::base_lco>
```

Public Static Functions

```
static constexpr lcos::base_lco *call (naming::address_type lva)
template<>
struct get_lva<lcos::base_lco const>
```

Public Static Functions

```
static constexpr lcos::base_lco const *call (naming::address_type lva)
namespace lcos
```

class base_lco

#include <base_lco.hpp> The base_lco class is the common base class for all LCO's implementing a simple set_event action

Subclassed by hpx::lcos::base_lco_with_value< Result, RemoteResult, ComponentTag >, hpx::lcos::base_lco_with_value< void, void, ComponentTag >

Public Types

```
typedef components::managed_component<base_lco> wrapping_type
typedef base_lco base_type_holder
```

Public Functions

```
virtual void set_event() = 0  
  
virtual void set_exception(std::exception_ptr const &e)  
  
virtual void connect(naming::id_type const&)  
  
virtual void disconnect(naming::id_type const&)  
  
virtual ~base_lco()  
    Destructor, needs to be virtual to allow for clean destruction of derived objects  
  
virtual void finalize()  
    finalize() will be called just before the instance gets destructed  
  
void set_event_nonvirt()  
    The function set_event_nonvirt is called whenever a set_event_action is applied on a instance of a LCO. This function just forwards to the virtual function set_event, which is overloaded by the derived concrete LCO.  
  
void set_exception_nonvirt(std::exception_ptr const &e)  
    The function set_exception is called whenever a set_exception_action is applied on a instance of a LCO. This function just forwards to the virtual function set_exception, which is overloaded by the derived concrete LCO.
```

Parameters

- e: [in] The exception encapsulating the error to report to this LCO instance.

```
void connect_nonvirt(naming::id_type const &id)
```

The function connect_nonvirt is called whenever a connect_action is applied on a instance of a LCO. This function just forwards to the virtual function connect, which is overloaded by the derived concrete LCO.

Parameters

- id: [in] target id

```
void disconnect_nonvirt(naming::id_type const &id)
```

The function disconnect_nonvirt is called whenever a disconnect_action is applied on a instance of a LCO. This function just forwards to the virtual function disconnect, which is overloaded by the derived concrete LCO.

Parameters

- id: [in] target id

```
HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco, set_event_nonvirt,  
                                    set_event_action)
```

Each of the exposed functions needs to be encapsulated into an action type, allowing to generate all required boilerplate code for threads, serialization, etc.

The set_event_action may be used to unconditionally trigger any LCO instances, it carries no additional parameters. The set_exception_action may be used to transfer arbitrary error information from the remote site to the LCO instance specified as a continuation. This action carries 2 parameters:

Parameters

- `std::exception_ptr`: [in] The exception encapsulating the error to report to this LCO instance.

```
set_exception_action HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco,      con-
                                         connect_nonvirt,      con-
                                         connect_action)
```

The *connect_action* may be used to.

The *set_exception_action* may be used to

Public Members

```
hpx::lcos::base_lco::set_exception_nonvirt  
set_exception_action disconnect_nonvirt
```

Public Static Functions

```
static components::component_type get_component_type()  
static void set_component_type(components::component_type type)
```

Defines

```
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION(...)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION(...)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION2(Value, RemoteValue, Name)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_1(Value)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_2(Value, Name)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_3(Value, RemoteValue, Name)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_4(Value, RemoteValue, Name, Tag)  
HPX_REGISTER_BASE_LCO_WITH_VALUE(...)  
HPX_REGISTER_BASE_LCO_WITH_VALUE(...)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_1(Value)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_2(Value, Name)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_3(Value, RemoteValue, Name)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_4(Value, RemoteValue, Name, Tag)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID(...)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID(...)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID2(Value, RemoteValue, Name, ActionIdGet, ActionIdSet)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_4(Value, Name, ActionIdGet, ActionIdSet)  
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_5(Value, RemoteValue, Name, ActionIdGet, ActionId-  
Set)
```

```
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_6 (Value, RemoteValue, Name, ActionIdGet, ActionIdSet,
                                         Tag)
```

```
namespace hpx
```

```
namespace lcos
```

```
template<typename Result, typename RemoteResult, typename ComponentTag>
class base_lco_with_value : public hpx::lcos::base_lco, public ComponentTag
    #include <base_lco_with_value.hpp> The base_lco_with_value class is the common base class for
    all LCO's synchronizing on a value. The RemoteResult template argument should be set to the type
    of the argument expected for the set_value action.
```

Template Parameters

- *RemoteResult*: The type of the result value to be carried back to the LCO instance.
- *ComponentTag*: The tag type representing the type of the component (either *component_tag* or *managed_component_tag*).

Public Types

```
template<>
using wrapping_type = typename detail::base_lco_wrapping_type<ComponentTag, base_lco_with_value>::ty
template<>
using base_type_holder = base_lco_with_value
```

Public Functions

```
void set_value_nonvirt (RemoteResult &&result)
```

The *function* *set_value_nonvirt* is called whenever a *set_value_action* is applied on this LCO instance. This function just forwards to the virtual function *set_value*, which is overloaded by the derived concrete LCO.

Parameters

- *result*: [in] The result value to be transferred from the remote operation back to this LCO instance.

```
Result get_value_nonvirt ()
```

The *function* *get_result_nonvirt* is called whenever a *get_result_action* is applied on this LCO instance. This function just forwards to the virtual function *get_result*, which is overloaded by the derived concrete LCO.

```
HPX_DEFINE_COMPONENT_DIRECT_ACTION (base_lco_with_value,      set_value_nonvirt,
                                    set_value_action)
```

The *set_value_action* may be used to trigger any LCO instances while carrying an additional parameter of any type.

RemoteResult is taken by rvalue ref. This allows for perfect forwarding. When the action thread function is created, the values are moved into the called function. If we took it by const lvalue reference, we would disable the possibility to further move the result to the designated destination.

Parameters

- `RemoteResult`: [in] The type of the result to be transferred back to this LCO instance. The `get_value_action` may be used to query the value this LCO instance exposes as its ‘result’ value.

Public Members

```
hpx::lcos::base_lco_with_value::get_value_nonvirt
```

Public Static Functions

```
static components::component_type get_component_type()
static void set_component_type(components::component_type type)
```

Protected Types

```
template<>
using result_type = std::conditional_t<std::is_void_v<Result>, util::unused_type, Result>
```

Protected Functions

```
~base_lco_with_value()
Destructor, needs to be virtual to allow for clean destruction of derived objects

void set_event()
void set_event_nonvirt(std::false_type)
void set_event_nonvirt(std::true_type)

virtual void set_value(RemoteResult &&result) = 0
virtual result_type get_value() = 0
virtual result_type get_value(error_code&)

template<typename ComponentTag>
class base_lco_with_value<void, void, ComponentTag> : public hpx::lcos::base_lco, public ComponentTag
#include <base_lco_with_value.hpp> The base_lco<void> specialization is used whenever the
set_event action for a particular LCO doesn't carry any argument.
```

Template Parameters

- `void`: This specialization expects no result value and is almost completely equivalent to the plain `base_lco`.

Public Types

```
template<>
using wrapping_type = typename detail::base_lco_wrapping_type<ComponentTag, base_lco_with_value>::ty
template<>
using base_type_holder = base_lco_with_value
template<>
using set_value_action = typename base_lco::set_event_action
```

Public Functions

```
void get_value()
```

Protected Functions

```
~base_lco_with_value()
```

Destructor, needs to be virtual to allow for clean destruction of derived objects

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename Archive, typename F, typename ...Ts>
```

```
void serialize(Archive &ar, ::hpx::util::detail::bound_action<F, Ts...> &bound, unsigned int  
const version = 0)
```

```
namespace util
```

Functions

```
template<typename Action, typename ...Ts, typename Enable = typename std::enable_if<traits::is_action<typename  
detail::bound_action<typename std::decay<Action>::type, typename util::make_index_pack<sizeof...(Ts)>::type, type>
```

```
template<typename Component, typename Signature, typename Derived, typename ...Ts>  
detail::bound_action<Derived, typename util::make_index_pack<sizeof...(Ts)>::type, typename std::decay<Ts>::type...>
```

```
template<>
struct typed_continuation<void, util::unused_type> : public hpx::actions::continuation
```

Public Types

```
template<>
using result_type = void
```

Public Functions

```
typed_continuation()
typed_continuation(naming::id_type const &id)
typed_continuation(naming::id_type &&id)

template<typename F>
typed_continuation(naming::id_type const &id, F &&f)

template<typename F>
typed_continuation(naming::id_type &&id, F &&f)

typed_continuation(naming::id_type const &id, naming::address &&addr)
typed_continuation(naming::id_type &&id, naming::address &&addr)

template<typename F>
typed_continuation(naming::id_type const &id, naming::address &&addr, F &&f)

template<typename F>
typed_continuation(naming::id_type &&id, naming::address &&addr, F &&f)

template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, typed_continuation>::value> F &&f)
typed_continuation(F &&f)

typed_continuation(typed_continuation&&)

typed_continuation &operator=(typed_continuation&&)

void trigger()

void trigger_value(util::unused_type&&)

void trigger_value(util::unused_type const&)
```

Private Types

```
template<>
using function_type = util::unique_function<void (naming::id_type)>
```

Private Functions

```
void serialize (hpx::serialization::input_archive &ar, unsigned)
void serialize (hpx::serialization::output_archive &ar, unsigned)
```

Private Members

```
function_type f_
```

Friends

```
friend hpx::serialization::access
    serialization support

namespace hpx

namespace actions

class continuation
    Subclassed by hpx::actions::typed_continuation< Result, Result >,
    hpx::actions::typed_continuation< void, util::unused_type >
```

Public Types

```
typedef void continuation_tag
```

Public Functions

```
continuation ()
continuation (naming::id_type const &id)
continuation (naming::id_type &&id)
continuation (naming::id_type const &id, naming::address &&addr)
continuation (naming::id_type &&id, naming::address &&addr)
continuation (continuation &&o)
continuation &operator= (continuation &&o)
void trigger_error (std::exception_ptr const &e)
void trigger_error (std::exception_ptr &&e)
void serialize (hpx::serialization::input_archive &ar, unsigned)
void serialize (hpx::serialization::output_archive &ar, unsigned)
constexpr naming::id_type const &get_id() const
constexpr naming::address get_addr() const
```

Protected Attributes

```
naming::id_type id_
naming::address addr_
```

```
template<typename Result, typename RemoteResult>
struct typed_continuation
```

Public Functions

```
typed_continuation()
typed_continuation(naming::id_type const &id)
typed_continuation(naming::id_type &&id)

template<typename F>
typed_continuation(naming::id_type const &id, F &&f)

template<typename F>
typed_continuation(naming::id_type &&id, F &&f)

typed_continuation(naming::id_type const &id, naming::address &&addr)
typed_continuation(naming::id_type &&id, naming::address &&addr)

template<typename F>
typed_continuation(naming::id_type const &id, naming::address &&addr, F &&f)

template<typename F>
typed_continuation(naming::id_type &&id, naming::address &&addr, F &&f)

template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, typed_continuation>(F &&f)
typed_continuation(typed_continuation&&)

typed_continuation &operator=(typed_continuation&&)
void trigger_value(RemoteResult &&result)
```

Private Types

```
template<>
using base_type = typed_continuation<RemoteResult>

template<>
using function_type = util::unique_function<void (naming::id_type, RemoteResult)>
```

Private Functions

```
template<typename Archive>
void serialize(Archive &ar, unsigned)
```

Friends

```
friend hpx::actions::hpx::serialization::access
    serialization support
```

```
template<typename Result>
struct typed_continuation<Result, Result> : public hpx::actions::continuation
```

Public Types

```
template<>
using result_type = Result
```

Public Functions

```
typed_continuation()
```

```
typed_continuation(naming::id_type const &id)
```

```
typed_continuation(naming::id_type &&id)
```

```
template<typename F>
typed_continuation(naming::id_type const &id, F &&f)
```

```
typed_continuation(naming::id_type &&id, F &&f)
```

```
typed_continuation(naming::id_type const &id, naming::address &&addr)
```

```
typed_continuation(naming::id_type &&id, naming::address &&addr)
```

```
template<typename F>
```

```
typed_continuation(naming::id_type const &id, naming::address &&addr, F &&f)
```

```
template<typename F>
```

```
typed_continuation(naming::id_type &&id, naming::address &&addr, F &&f)
```

```
template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, F>>::type>
typed_continuation(F &&f)
```

```
typed_continuation(typed_continuation&&)
```

```
typed_continuation &operator=(typed_continuation&&)
```

```
void trigger_value(Result &&result)
```

Protected Attributes

```
function_type f_
```

Private Types

```
template<>
using function_type = util::unique_function<void (naming::id_type, Result)>
```

Private Functions

```
template<typename Archive>
void serialize(Archive &ar, unsigned)
```

Friends

```
friend hpx::actions::hpx::serialization::access
    serialization support
```

```
template<>
struct typed_continuation<void, util::unused_type> : public hpx::actions::continuation
```

Public Types

```
template<>
using result_type = void
```

Public Functions

```
typed_continuation()
```

```
typed_continuation(naming::id_type const &id)
```

```
typed_continuation(naming::id_type &&id)
```

```
template<typename F>
```

```
typed_continuation(naming::id_type const &id, F &&f)
```

```
template<typename F>
```

```
typed_continuation(naming::id_type &&id, F &&f)
```

```
typed_continuation(naming::id_type const &id, naming::address &&addr)
```

```
typed_continuation(naming::id_type &&id, naming::address &&addr)
```

```
template<typename F>
```

```
typed_continuation(naming::id_type const &id, naming::address &&addr, F &&f)
```

```
template<typename F>
```

```
typed_continuation(naming::id_type &&id, naming::address &&addr, F &&f)
```

```
template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type,
```

```
typed_continuation(F &&f)
```

```
typed_continuation (typed_continuation&&)
typed_continuation &operator= (typed_continuation&&)
void trigger ()
void trigger_value (util::unused_type&&)
void trigger_value (util::unused_type const&)
```

Private Types

```
template<>
using function_type = util::unique_function<void (naming::id_type)>
```

Private Functions

```
void serialize (hpx::serialization::input_archive &ar, unsigned)
void serialize (hpx::serialization::output_archive &ar, unsigned)
```

Private Members

```
function_type f_
```

Friends

```
friend hpx::actions::hpx::serialization::access
    serialization support

namespace hpx

namespace actions
```

```
template<typename Cont, typename F>
struct continuation2_impl
```

Public Functions

```
continuation2_impl ()
template<typename Cont_, typename F_>
continuation2_impl (Cont_ &&cont, hpx::id_type const &target, F_ &&f)
virtual ~continuation2_impl ()
template<typename T>
```

util::invoke_result<function_type, hpx::id_type, typename util::invoke_result<cont_type, hpx::id_type, T>::type>::ty

Private Types

```
template<>
using cont_type = typename std::decay<Cont>::type
template<>
using function_type = typename std::decay<F>::type
```

Private Functions

```
template<typename Archive>
void serialize(Archive &ar, unsigned int const)
```

Private Members

```
cont_type cont_
hpx::id_type target_
function_type f_
```

Friends

```
friend hpx::actions::hpx::serialization::access
namespace hpx

namespace actions
```

Functions

```
template<typename Result, typename RemoteResult, typename F, typename ...Ts>
void trigger(typed_continuation<Result, RemoteResult>&&, F&&, Ts&&...)
namespace hpx

namespace actions

template<typename Cont>
struct continuation_impl
```

Public Functions

```
continuation_impl()  
  
template<typename Cont_>  
continuation_impl(Cont_ &&cont, hpx::id_type const &target)  
  
virtual ~continuation_impl()  
  
template<typename T>  
util::invoke_result<cont_type, hpx::id_type, T>::type operator()(hpx::id_type      const  
                                         &lco, T &&t) const
```

Private Types

```
template<>  
using cont_type = typename std::decay<Cont>::type
```

Private Functions

```
template<typename Archive>  
void serialize(Archive &ar, unsigned int const)
```

Private Members

```
cont_type cont_  
hpx::id_type target_
```

Friends

```
friend hpx::actions::hpx::serialization::access  
namespace hpx
```

Functions

```
template<typename Action, typename T0, typename ...Ts, typename Enable = typename std::enable_if<traits::is_action</>
```

```
auto dataflow(T0 &&t0, Ts&&... ts)
```

```
HPX_FORWARD(T0, t0)
```

```
HPX_FORWARD(Ts, ts)
```

```
template<typename Action, typename Allocator, typename T0, typename ...Ts, typename Enable = typename std::enable_if<traits::is_allocator</>
```

```
auto dataflow_alloc(Allocator const &alloc, T0 &&t0, Ts&&... ts)
```

```
namespace hpx
```

```
namespace lcos
```

```
template<typename Action, typename Result = typename traits::promise_local_result<typename Action::remote_r...
```

class packaged_action

`#include <packaged_action.hpp>` A *packaged_action* can be used by a single *thread* to invoke a (remote) action and wait for the result. The result is expected to be sent back to the *packaged_action* using the LCO's `set_event` action

A *packaged_action* is one of the simplest synchronization primitives provided by HPX. It allows to synchronize on a eager evaluated remote operation returning a result of the type *Result*.

Note The action executed using the *packaged_action* as a continuation must return a value of a type convertible to the type as specified by the template parameter *Result*.

Template Parameters

- *Action*: The template parameter *Action* defines the action to be executed by this *packaged_action* instance. The arguments *arg0*, ..., *argN* are used as parameters for this action.
- *Result*: The template parameter *Result* defines the type this *packaged_action* is expected to return from its associated future `packaged_action::get_future`.
- *DirectExecute*: The template parameter *DirectExecute* is an optimization aid allowing to execute the action directly if the target is local (without spawning a new thread for this). This template does not have to be supplied explicitly as it is derived from the template parameter *Action*.

```
template<typename Result, typename RemoteResult>
class promise
```

`#include <promise.hpp>` A promise can be used by a single *thread* to invoke a (remote) action and wait for the result. The result is expected to be sent back to the promise using the LCO's `set_event` action

A promise is one of the simplest synchronization primitives provided by HPX. It allows to synchronize on a eager evaluated remote operation returning a result of the type *Result*. The *promise* allows to synchronize exactly one *thread* (the one passed during construction time).

```
// Create the promise (the expected result is a id_type)
lcos::promise<naming::id_type> p;

// Get the associated future
future<naming::id_type> f = p.get_future();

// initiate the action supplying the promise as a
// continuation
apply<some_action>(*new continuation(p.get_id()), ...);

// Wait for the result to be returned, yielding control
// in the meantime.
naming::id_type result = f.get();
// ...
```

Note The action executed by the promise must return a value of a type convertible to the type as specified by the template parameter *RemoteResult*

Template Parameters

- *Result*: The template parameter *Result* defines the type this promise is expected to return from `promise::get`.
- *RemoteResult*: The template parameter *RemoteResult* defines the type this promise is expected to receive from the remote action.

```
namespace hpx
```

Functions

```
hpx::actions::set_lco_value_continuation make_continuation()
```

```
template<typename Cont>  
hpx::actions::continuation_impl<typename std::decay<Cont>::type> make_continuation(Cont  
&&cont)
```

```
template<typename Cont>  
hpx::actions::continuation_impl<typename std::decay<Cont>::type> make_continuation(Cont  
&&f,  
hpx::id_type  
const  
&tar-  
get)
```

```
template<typename Cont, typename F>  
std::enable_if<!std::is_same<typename std::decay<F>::type, hpx::id_type>::value, hpx::actions::continuation2_impl<typena
```

```
template<typename Cont, typename F>  
hpx::actions::continuation2_impl<typename std::decay<Cont>::type, typename std::decay<F>::type> make_continuation
```

```
namespace hpx
```

```
namespace lcos
```

```
template<typename Action, typename Result>  
class packaged_action<Action, Result, false> : public hpx::lcos::promise<Result, hpx::traits::extract_action<Ac
```

Subclassed by *hpx::lcos::packaged_action< Action, Result, true >*

Public Functions

```
packaged_action()
```

```
template<typename Allocator>  
packaged_action(std::allocator_arg_t, Allocator const &alloc)
```

```
template<typename ...Ts>  
void apply(naming::id_type const &id, Ts&&... vs)
```

```
template<typename ...Ts>  
void apply(naming::address &&addr, naming::id_type const &id, Ts&&... vs)
```

```

template<typename Callback, typename ...Ts>
void apply_cb(naming::id_type const &id, Callback &&cb, Ts&&... vs)
```

```

template<typename Callback, typename ...Ts>
void apply_cb(naming::address &&addr, naming::id_type const &id, Callback &&cb,
Ts&&... vs)
```

```

template<typename ...Ts>
void apply_p(naming::id_type const &id, threads::thread_priority priority, Ts&&... vs)
```

```

template<typename ...Ts>
void apply_p(naming::address &&addr, naming::id_type const &id,
threads::thread_priority priority, Ts&&... vs)
```

```

template<typename Callback, typename ...Ts>
void apply_p_cb(naming::id_type const &id, threads::thread_priority priority, Callback
&&cb, Ts&&... vs)
```

```

template<typename Callback, typename ...Ts>
void apply_p_cb(naming::address &&addr, naming::id_type const &id, Callback &&cb, Ts&&... vs)
```

```

template<typename ...Ts>
void apply_deferred(naming::address &&addr, naming::id_type const &id, Ts&&... vs)
```

```

template<typename Callback, typename ...Ts>
void apply_deferred_cb(naming::address &&addr, naming::id_type const &id, Callback &&cb, Ts&&... vs)
```

Protected Types

```

template<>
using action_type = typename hpx::traits::extract_action<Action>::type
```

```

template<>
using remote_result_type = typename action_type::remote_result_type
```

```

template<>
using base_type = promise<Result, remote_result_type>
```

Protected Functions

```

template<typename ...Ts>
void do_apply(naming::address &&addr, naming::id_type const &id,
threads::thread_priority priority, Ts&&... vs)
```

```

template<typename ...Ts>
void do_apply(naming::id_type const &id, threads::thread_priority priority, Ts&&... vs)
```

```

template<typename Callback, typename ...Ts>
void do_apply_cb(naming::address &&addr, naming::id_type const &id,
threads::thread_priority priority, Callback &&cb, Ts&&... vs)
```

```

template<typename Callback, typename ...Ts>
void do_apply_cb(naming::id_type const &id, threads::thread_priority priority, Callback
&&cb, Ts&&... vs)
```

```

template<typename Action, typename Result>
class packaged_action<Action, Result, true> : public hpx::lcos::packaged_action<Action, Result, false>
```

Public Functions

```
packaged_action()
Construct a (non-functional) instance of an packaged_action. To use this instance its member
function apply needs to be directly called.

template<typename Allocator>
packaged_action(std::allocator_arg_t, Allocator const &alloc)

template<typename ...Ts>
void apply(naming::id_type const &id, Ts&&... vs)

template<typename ...Ts>
void apply(naming::address &&addr, naming::id_type const &id, Ts&&... vs)

template<typename Callback, typename ...Ts>
void apply_cb(naming::id_type const &id, Callback &&cb, Ts&&... vs)

template<typename Callback, typename ...Ts>
void apply_cb(naming::address &&addr, naming::id_type const &id, Callback &&cb,
              Ts&&... vs)
```

Private Types

```
template<>
using action_type = typename packaged_action::action_type

template<>
class promise<void, hpx::util::unused_type> : public detail::promise_base<void, hpx::util::unused_type, detail::promise_data<
```

Public Functions

```
promise()
constructs a promise object and a shared state.

template<typename Allocator>
promise(std::allocator_arg_t, Allocator const &a)
constructs a promise object and a shared state. The constructor uses the allocator a to allocate the memory
for the shared state.

promise(promise &&other)
constructs a new promise object and transfers ownership of the shared state of other (if any) to the newly-
constructed object.
```

Post other has no shared state.

```
~promise()
Abandons any shared state.

promise &operator=(promise &&other)
Abandons any shared state (30.6.4) and then as if promise(HPX_MOVE(other)).swap(*this).
```

Return *this.

```
void swap (promise &other)
    Exchanges the shared state of *this and other.
```

Post *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

```
void set_value ()
    atomically stores the value r in the shared state and makes that state ready (30.6.4).
```

Exceptions

- `future_error`: if its shared state already has a stored value. if shared state has no stored value exception is raised. `promise_already_satisfied` if its shared state already has a stored value or exception. `no_state` if *this has no shared state.

Private Types

```
template<>
using base_type = detail::promise_base<void, hpx::util::unused_type, detail::promise_data<void>>

template<typename R, typename Allocator>
struct uses_allocator<hpx::lcos::promise<R>, Allocator> : public true_type
    #include <promise.hpp> Requires: Allocator shall be an allocator (17.6.3.5)

namespace hpx

namespace lcos
```

Functions

```
template<typename Result, typename RemoteResult>
void swap (promise<Result, RemoteResult> &x, promise<Result, RemoteResult> &y)
```

```
template<>
class promise<void, hpx::util::unused_type> : public detail::promise_base<void, hpx::util::unused_type, detail::promise_data<void>>
```

Public Functions

```
promise ()
    constructs a promise object and a shared state.
```

```
template<typename Allocator>
promise (std::allocator_arg_t, Allocator const &a)
    constructs a promise object and a shared state. The constructor uses the allocator a to allocate the memory for the shared state.
```

```
promise (promise &&other)
    constructs a new promise object and transfers ownership of the shared state of other (if any) to the newly- constructed object.
```

Post other has no shared state.

~promise()

Abandons any shared state.

promise &operator=(promise &&other)

Abandons any shared state (30.6.4) and then as if promise(HPX_MOVE(other)).swap(*this).

Return *this.

void swap(promise &other)

Exchanges the shared state of *this and other.

Post *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

void set_value()

atomically stores the value r in the shared state and makes that state ready (30.6.4).

Exceptions

- `future_error`: if its shared state already has a stored value. if shared state has no stored value exception is raised. `promise_already_satisfied` if its shared state already has a stored value or exception. `no_state` if *this has no shared state.

Private Types

```
template<>
using base_type = detail::promise_base<void, hpx::util::unused_type, detail::promise_data<void>>

namespace std

template<typename R, typename Allocator>
struct uses_allocator<hpx::lcos::promise<R>, Allocator> : public true_type
#include <promise.hpp> Requires: Allocator shall be an allocator (17.6.3.5)

namespace hpx

namespace actions

struct set_lco_value_continuation
```

Public Functions**template<typename T>**

T operator() (*naming::id_type const &lco, T &&t*) **const**

struct set_lco_value_unmanaged_continuation

Public Functions

```
template<typename T>
T operator() (naming::id_type const &lco, T &&t) const

namespace hpx
```

Functions

```
template<typename Action, typename F, typename ...Ts>
auto sync (F &&f, Ts&&... ts)

namespace hpx
```

```
namespace actions
```

Functions

```
template<typename Result, typename RemoteResult, typename F, typename ...Ts>
void trigger (typed_continuation<Result, RemoteResult> &&cont, F &&f, Ts&&... vs)

template<typename Result, typename F, typename ...Ts>
void trigger (typed_continuation<Result, util::unused_type> &&cont, F &&f, Ts&&... vs)

namespace hpx
```

Functions

```
void trigger_lco_event (naming::id_type const &id, naming::address &&addr, bool
move_credits = true)
Trigger the LCO referenced by the given id.
```

Parameters

- `id`: [in] This represents the id of the LCO which should be triggered.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `move_credits`: [in] If this is set to `true` then it is ok to send all credits in `id` along with the generated message. The default value is `true`.

```
void trigger_lco_event (naming::id_type const &id, bool move_credits = true)
Trigger the LCO referenced by the given id.
```

Parameters

- `id`: [in] This represents the id of the LCO which should be triggered.
- `move_credits`: [in] If this is set to `true` then it is ok to send all credits in `id` along with the generated message. The default value is `true`.

```
void trigger_lco_event(naming::id_type const &id, naming::address &&addr, naming::id_type
                           const &cont, bool move_credits = true)
Trigger the LCO referenced by the given id.
```

Parameters

- `id`: [in] This represents the id of the LCO which should be triggered.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in `id` along with the generated message. The default value is *true*.

```
void trigger_lco_event(naming::id_type const &id, naming::id_type const &cont, bool
                           move_credits = true)
Trigger the LCO referenced by the given id.
```

Parameters

- `id`: [in] This represents the id of the LCO which should be triggered.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in `id` along with the generated message. The default value is *true*.

```
template<typename Result>
void set_lco_value(naming::id_type const &id, naming::address &&addr, Result &&t, bool
                           move_credits = true)
Set the result value for the LCO referenced by the given id.
```

Parameters

- `id`: [in] This represents the id of the LCO which should receive the given value.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `t`: [in] This is the value which should be sent to the LCO.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in `id` along with the generated message. The default value is *true*.

```
template<typename Result>
std::enable_if<!std::is_same<typename std::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 std::decay<Result>::type, naming::address>::value>::type set_lco_value_unman
```

Set the result value for the (unmanaged) LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the given value.
- `t`: [in] This is the value which should be sent to the LCO.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

```
template<typename Result>
void set_lco_value(naming::id_type const &id, naming::address &&addr, Result &&t, naming::id_type const &cont, bool move_credits = true)
```

Set the result value for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the given value.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `t`: [in] This is the value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

```
template<typename Result>
```

```
std::enable_if<!std::is_same<typename std::decay<Result>::type, naming::address>::value>::type set_lco_value(naming::address const &id, Result &&t, naming::id 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 std::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*.

namespace hpx

Functions

```

template<typename Action, typename ...Ts>
bool apply_p(naming::id_type const &id, threads::thread_priority priority, Ts&&... vs)
```

```

template<typename Action, typename Client, typename Stub, typename ...Ts>
bool apply_p(components::client_base<Client, Stub> const &c, threads::thread_priority priority,
              Ts&&... vs)
```

```

template<typename Action, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_p(DistPolicy
                                         const &policy,
                                         threads::thread_priority
                                         priority,
                                         Ts&&... vs)
```

```

template<typename Action, typename ...Ts>
bool apply(naming::id_type const &id, Ts&&... vs)
```

```

template<typename Action, typename Client, typename Stub, typename ...Ts>
bool apply(components::client_base<Client, Stub> const &c, Ts&&... vs)
```

```

template<typename Action, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply(DistPolicy const
                                         &policy, Ts&&...
                                         vs)
```

```

template<typename Action, typename Continuation, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply_p(Continuation &&c,
                                         naming::id_type
                                         const &gid,
                                         threads::thread_priority
                                         priority, Ts&&...
                                         vs)
```

```

template<typename Action, typename Continuation, typename Client, typename Stub, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply_p(Continuation
                                         &&cont, components::client_base<Client,
                                         Stub> const &c,
                                         threads::thread_priority
                                         priority, Ts&&...
                                         vs)
```

```

template<typename Action, typename Continuation, typename DistPolicy, typename ...Ts>
```

```
std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply(Continuation &&c, naming::id_type const &gid, Ts&&... vs)
```

```
template<typename Action, typename Continuation, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply(Continuation &&cont, components::client_base<Client, Stub> const &c, Ts&&... vs)
```

```
template<typename Action, typename Continuation, typename Client, typename Stub, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply(Continuation &&cont, components::client_base<Client, Stub> const &c, Ts&&... vs)
```

```
template<typename Action, typename Continuation, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value && traits::is_continuation<Continuation>::value, bool>::type apply_c_p(naming::id_type const &contgid, naming::id_type const &gid, threads::thread_priority priority, Ts&&... vs)
```

```
template<typename Action, typename ...Ts>
bool apply_c(naming::id_type const &contgid, naming::id_type const &gid, Ts&&... vs)
```

```
template<typename Component, typename Signature, typename Derived, typename ...Ts>
bool apply_c(hpx::actions::basic_action<Component, Signature, Derived>, naming::id_type const &contgid, naming::id_type const &gid, Ts&&... vs)
```

```
namespace hpx
```

Functions

```

template<typename Action, typename Callback, typename ...Ts>
bool apply_p_cb(naming::id_type const &gid, threads::thread_priority priority, Callback &&cb,
                  Ts&&... vs)

template<typename Action, typename Callback, typename ...Ts>
bool apply_cb(naming::id_type const &gid, Callback &&cb, Ts&&... vs)

template<typename Component, typename Signature, typename Derived, typename Callback, typename ...Ts>
bool apply_cb(hpx::actions::basic_action<Component, Signature, Derived>, naming::id_type const
               &gid, Callback &&cb, Ts&&... vs)

template<typename Action, typename DistPolicy, typename Callback, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_p_cb(DistPolicy
                                         const
                                         &policy,
                                         threads::thread_priority
                                         priority,
                                         Callback
                                         &&cb,
                                         Ts&&...
                                         vs)

template<typename Action, typename DistPolicy, typename Callback, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_cb(DistPolicy
                                         const &pol-
                                         icy,      Call-
                                         back   &&cb,
                                         Ts&&... vs)

template<typename Component, typename Signature, typename Derived, typename DistPolicy, typename Callback,
         std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_cb(hpx::actions::basic_action<Component
                                         Signature, De-
                                         rived>, Dist-
                                         Policy const
                                         &policy, Call-
                                         back   &&cb,
                                         Ts&&... vs)

template<typename Action, typename Continuation, typename Callback, typename ...Ts>
bool apply_p_cb(Continuation &&c, naming::address &&addr, naming::id_type const &gid,
                  threads::thread_priority priority, Callback &&cb, Ts&&... vs)

template<typename Action, typename Continuation, typename Callback, typename ...Ts>
bool apply_p_cb(Continuation &&c, naming::id_type const &gid, threads::thread_priority priority,
                  Callback &&cb, Ts&&... vs)

template<typename Action, typename Continuation, typename Callback, typename ...Ts>
bool apply_cb(Continuation &&c, naming::id_type const &gid, Callback &&cb, Ts&&... vs)

template<typename Component, typename Continuation, typename Signature, typename Derived, typename Callback,
         std::enable_if<traits::is_distribution_policy<naming::id_type const &gid>::value, bool>::type apply_cb(Continuation &&c, hpx::actions::basic_action<Component, Signature, Derived>, nam-
                                         ing::id_type const &gid, Callback &&cb, Ts&&... vs)

template<typename Action, typename Continuation, typename DistPolicy, typename Callback, typename ...Ts>
```

```
std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_c_p_cb(Continuation &&c, hpx::actions::basic_action<Component, Signature, Derived>, DistPolicy const &policy, Callback &&cb, Ts&&... vs)
```

```
template<typename Action, typename Continuation, typename DistPolicy, typename Callback, typename ...Ts> std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_c_p_cb(naming::id_type const &contgid, naming::id_type const &gid, threads::thread_priority priority, Callback &&cb, Ts&&... vs)
```

```
template<typename Action, typename Continuation, typename DistPolicy, typename Callback, typename ...Ts> std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_c_p_cb(naming::id_type const &contgid, naming::id_type const &gid, Callback &&cb, Ts&&... vs)
```

```
template<typename Action, typename Continuation, typename DistPolicy, typename Callback, typename ...Ts> std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_c_p_cb(naming::id_type const &contgid, naming::address &&addr, naming::id_type const &gid, threads::thread_priority priority, Callback &&cb, Ts&&... vs)
```

```
template<typename Action, typename Continuation, typename DistPolicy, typename Callback, typename ...Ts> std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_c_p_cb(naming::id_type const &contgid, naming::id_type const &gid, threads::thread_priority priority, Callback &&cb, Ts&&... vs)
```

```
bool apply_c_cb(naming::id_type const &contgid, naming::address &&addr, naming::id_type
const &gid, Callback &&cb, Ts&&... vs)

namespace functional
```

Functions

```
template<typename Action, typename Callback, typename ...Ts>
apply_c_p_cb_impl<Action, typename std::decay<Callback>::type, typename std::decay<Ts>::type...> apply_c_p_c
```

```
template<typename Action, typename Callback, typename ...Ts>
struct apply_c_p_cb_impl
```

Public Types

```
typedef hpx::tuple<Ts...> tuple_type
```

Public Functions

```
template<typename ...Ts_>
apply_c_p_cb_impl(naming::id_type const &contid, naming::address &&addr, naming::id_type const &id, threads::thread_priority p, Callback &&cb,
Ts_&&... vs)

apply_c_p_cb_impl(apply_c_p_cb_impl &&rhs)
apply_c_p_cb_impl &operator=(apply_c_p_cb_impl &&rhs)
void operator() ()
```

Protected Functions

```
template<std::size_t... Is>
void apply_action(util::index_pack<Is...>)
```

Private Members

```
naming::id_type contid_
naming::address addr_
naming::id_type id_
threads::thread_priority p_
Callback cb_
tuple_type args_
```

```
namespace hpx
```

Functions

```
template<typename Action, typename Cont, typename ...Ts>
bool apply_continue(Cont &&cont, id_type const &gid, Ts&&... vs)
```

```
template<typename Component, typename Signature, typename Derived, typename Cont, typename ...Ts>
bool apply_continue(hpx::actions::basic_action<Component, Signature, Derived>, Cont &&cont,
                   id_type const &gid, Ts&&... vs)
```

```
template<typename Action, typename ...Ts>
bool apply_continue(id_type const &cont, id_type const &gid, Ts&&... vs)
```

```
template<typename Component, typename Signature, typename Derived, typename ...Ts>
bool apply_continue(hpx::actions::basic_action<Component, Signature, Derived>, id_type const
                   &cont, id_type const &gid, Ts&&... vs)
```

```
namespace hpx
```

Functions

```
template<typename Action, typename Cont, typename Callback, typename ...Ts>
bool apply_continue_cb(Cont &&cont, id_type const &gid, Callback &&cb, Ts&&... vs)
```

```
template<typename Component, typename Signature, typename Derived, typename Cont, typename Callback, typename ...Ts>
bool apply_continue_cb(hpx::actions::basic_action<Component, Signature, Derived>, Cont
                      &&cont, id_type const &gid, Callback &&cb, Ts&&... vs)
```

```
template<typename Action, typename Callback, typename ...Ts>
bool apply_continue_cb(id_type const &cont, id_type const &gid, Callback &&cb, Ts&&...
                      vs)
```

```
template<typename Component, typename Signature, typename Derived, typename Callback, typename ...Ts>
bool apply_continue_cb(hpx::actions::basic_action<Component, Signature, Derived>, id_type const
                      &cont, id_type const &gid, Callback &&cb, Ts&&... vs)
```

```
namespace hpx
```

Functions

```
template<typename Action, typename Cont, typename ...Ts>
bool apply_continue (Cont &&cont, naming::id_type const &gid, Ts&&... vs)

template<typename Component, typename Signature, typename Derived, typename Cont, typename ...Ts>
bool apply_continue (hpx::actions::basic_action<Component, Signature, Derived>, Cont &&cont,
                     naming::id_type const &gid, Ts&&... vs)

template<typename Action, typename ...Ts>
bool apply_continue (naming::id_type const &cont, naming::id_type const &gid, Ts&&... vs)

template<typename Component, typename Signature, typename Derived, typename ...Ts>
bool apply_continue (hpx::actions::basic_action<Component, Signature, Derived>, naming::id_type const &cont, naming::id_type const &gid, Ts&&... vs)

namespace hpx
```

```
namespace applier
```

Functions

```
template<typename Arg0>
void trigger (naming::id_type const &k, Arg0 &&arg0)

void trigger (naming::id_type const &k)

void trigger_error (naming::id_type const &k, std::exception_ptr const &e)

void trigger_error (naming::id_type const &k, std::exception_ptr &&e)

template<typename Result, typename RemoteResult>
struct action_trigger_continuation<actions::typed_continuation<Result, RemoteResult>>
```

Public Static Functions

```
template<typename F, typename ...Ts>
static void call (actions::typed_continuation<Result, RemoteResult> &&cont, F &&f, Ts&&... ts)

namespace hpx
```

```
namespace traits
```

```
template<typename Result, typename RemoteResult>
struct action_trigger_continuation<actions::typed_continuation<Result, RemoteResult>>
```

Public Static Functions

```
template<typename F, typename ...Ts>
static void call (actions::typed_continuation<Result, RemoteResult> &&cont, F &&f,
                  Ts&&... ts)
```

checkpoint

See *Public API* for a list of names and headers that are part of the public HPX API.

This header defines the save_checkpoint and restore_checkpoint functions. These functions are designed to help HPX application developer's checkpoint their applications. Save_checkpoint serializes one or more objects and saves them as a byte stream. Restore_checkpoint converts the byte stream back into instances of the objects.

```
namespace hpx
```

```
namespace util
```

Functions

```
std::ostream &operator<< (std::ostream &ost, checkpoint const &ckp)
    Operator<< Overload
```

This overload is the main way to write data from a checkpoint to an object such as a file. Inside the function, the size of the checkpoint will be written to the stream before the checkpoint's data. The operator>> overload uses this to read the correct number of bytes. Be mindful of this additional write and read when you use different facilities to write out or read in data to a checkpoint!

Parameters

- ost: Output stream to write to.
- ckp: Checkpoint to copy from.

Return Operator<< returns the ostream object.

```
std::istream &operator>> (std::istream &ist, checkpoint &ckp)
    Operator>> Overload
```

This overload is the main way to read in data from an object such as a file to a checkpoint. It is important to note that inside the function, the first variable to be read is the size of the checkpoint. This size variable is written to the stream before the checkpoint's data in the operator<< overload. Be mindful of this additional read and write when you use different facilities to read in or write out data from a checkpoint!

Parameters

- ist: Input stream to write from.
- ckp: Checkpoint to write to.

Return Operator>> returns the ostream object.

```
template<typename T, typename ...Ts, typename U = typename std::enable_if<!hpx::traits::is_launch_policy<T>::value &
          hpx::future<checkpoint> save_checkpoint (T &&t, Ts&&... ts)
            Save_checkpoint
```

`Save_checkpoint` takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a `shared_ptr` to the component or by passing a component's client instance to `save_checkpoint`. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used `save_checkpoint` will simply return a checkpoint object.

Template Parameters

- `T`: Containers passed to `save_checkpoint` to be serialized and placed into a checkpoint object.
- `Ts`: More containers passed to `save_checkpoint` to be serialized and placed into a checkpoint object.
- `U`: This parameter is used to make sure that `T` is not a launch policy or a checkpoint. This forces the compiler to choose the correct overload.

Parameters

- `t`: A container to restore.
- `ts`: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Return `Save_checkpoint` returns a future to a checkpoint with one exception: if you pass `hpx::launch::sync` as the first argument. In this case `save_checkpoint` will simply return a checkpoint.

```
template<typename T, typename ...Ts>
hpx::future<checkpoint> save_checkpoint (checkpoint &&c, T &&t, Ts&&... ts)
    Save_checkpoint - Take a pre-initialized checkpoint
```

`Save_checkpoint` takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a `shared_ptr` to the component or by passing a component's client instance to `save_checkpoint`. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used `save_checkpoint` will simply return a checkpoint object.

Template Parameters

- `T`: Containers passed to `save_checkpoint` to be serialized and placed into a checkpoint object.
- `Ts`: More containers passed to `save_checkpoint` to be serialized and placed into a checkpoint object.

Parameters

- `c`: Takes a pre-initialized checkpoint to copy data into.
- `t`: A container to restore.
- `ts`: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Return `Save_checkpoint` returns a future to a checkpoint with one exception: if you pass `hpx::launch::sync` as the first argument. In this case `save_checkpoint` will simply return a checkpoint.

```
template<typename T, typename ...Ts, typename U = typename std::enable_if<!std::is_same<typename std::decay<T>
hpx::future<checkpoint> save_checkpoint (hpx::launch p, T &&t, Ts&&... ts)
    Save_checkpoint - Policy overload
```

`Save_checkpoint` takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a `shared_ptr` to the component or by passing a component's client instance to `save_checkpoint`. Additionally the function

can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- T: Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- Ts: More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- p: Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- t: A container to restore.
- ts: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Return Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a checkpoint.

```
template<typename T, typename ...Ts>
hpx::future<checkpoint> save_checkpoint (hpx::launch p, checkpoint &&c, T &&t, Ts&&...
                                         ts)
Save_checkpoint - Policy overload & pre-initialized checkpoint
```

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- T: Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- Ts: More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- p: Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- c: Takes a pre-initialized checkpoint to copy data into.
- t: A container to restore.
- ts: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Return Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a checkpoint.

```
template<typename T, typename ...Ts, typename U = typename std::enable_if<!std::is_same<typename std::decay<T>
checkpoint save_checkpoint (hpx::launch::sync_policy sync_p, T &&t, Ts&&... ts)
Save_checkpoint - Sync_policy overload
```

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- T: Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- Ts: More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- U: This parameter is used to make sure that T is not a checkpoint. This forces the compiler to choose the correct overload.

Parameters

- sync_p: hpx::launch::sync_policy
- t: A container to restore.
- ts: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Return Save_checkpoint which is passed hpx::launch::sync_policy will return a checkpoint which contains the serialized values checkpoint.

```
template<typename T, typename ...Ts>
checkpoint save_checkpoint (hpx::launch::sync_policy sync_p, checkpoint &&c, T &&t,
                        Ts&&... ts)
Save_checkpoint - Sync_policy overload & pre-init. checkpoint
```

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- T: Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- Ts: More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- sync_p: hpx::launch::sync_policy
- c: Takes a pre-initialized checkpoint to copy data into.
- t: A container to restore.
- ts: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Return Save_checkpoint which is passed hpx::launch::sync_policy will return a checkpoint which contains the serialized values checkpoint.

```
template<typename T, typename ...Ts, typename U = typename std::enable_if<!hpx::traits::is_launch_policy<T>::value &
hpx::future<checkpoint> prepare_checkpoint (T const &t, Ts const&... ts)
prepare_checkpoint
```

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Return prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

Template Parameters

- T: A container to restore.
- Ts: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- t: A container to restore.

- `ts`: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

```
template<typename T, typename ...Ts>
hpx::future<checkpoint> prepare_checkpoint (checkpoint &&c, T const &t, Ts const&...
                                         ts)
prepare_checkpoint
```

`prepare_checkpoint` takes the containers which have to be filled from the byte stream by a subsequent `restore_checkpoint` invocation. `prepare_checkpoint` will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Return `prepare_checkpoint` returns a properly resized checkpoint object that can be used for a subsequent `restore_checkpoint` operation.

Template Parameters

- `T`: A container to restore.
- `Ts`: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- `c`: Takes a pre-initialized checkpoint to prepare
- `t`: A container to restore.
- `ts`: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

```
template<typename T, typename ...Ts, typename U = typename std::enable_if<!std::is_same<T, checkpoint>::value>::type>
hpx::future<checkpoint> prepare_checkpoint (hpx::launch p, T const &t, Ts const&... ts)
prepare_checkpoint
```

`prepare_checkpoint` takes the containers which have to be filled from the byte stream by a subsequent `restore_checkpoint` invocation. `prepare_checkpoint` will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Return `prepare_checkpoint` returns a properly resized checkpoint object that can be used for a subsequent `restore_checkpoint` operation.

Template Parameters

- `T`: A container to restore.
- `Ts`: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- `p`: Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. `async`, `sync`, etc.
- `t`: A container to restore.
- `ts`: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

```
template<typename T, typename ...Ts>
hpx::future<checkpoint> prepare_checkpoint (hpx::launch p, checkpoint &&c, T const &t,
                                         Ts const&... ts)
prepare_checkpoint
```

`prepare_checkpoint` takes the containers which have to be filled from the byte stream by a subsequent `restore_checkpoint` invocation. `prepare_checkpoint` will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Return prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

Template Parameters

- T: A container to restore.
- Ts: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- p: Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- c: Takes a pre-initialized checkpoint to prepare
- t: A container to restore.
- ts: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

```
template<typename T, typename ...Ts>
void restore_checkpoint(checkpoint const &c, T &t, Ts&... ts)
    Restore_checkpoint
```

Restore_checkpoint takes a checkpoint object as a first argument and the containers which will be filled from the byte stream (in the same order as they were placed in save_checkpoint). Restore_checkpoint can resurrect a stored component in two ways: by passing in a instance of a component's shared_ptr or by passing in an instance of the component's client.

Return Restore_checkpoint returns void.

Template Parameters

- T: A container to restore.
- Ts: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- c: The checkpoint to restore.
- t: A container to restore.
- ts: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

```
class checkpoint
#include <checkpoint.hpp> Checkpoint Object
```

Checkpoint is the container object which is produced by save_checkpoint and is consumed by a restore_checkpoint. A checkpoint may be moved into the save_checkpoint object to write the byte stream to the pre-created checkpoint object.

Checkpoints are able to store all containers which are able to be serialized including components.

Public Types

```
using const_iterator = std::vector::const_iterator
```

Public Functions

```
checkpoint()
~checkpoint()

checkpoint(checkpoint const &c)
checkpoint(checkpoint &&c)

checkpoint(std::vector<char> const &vec)
checkpoint(std::vector<char> &&vec)

checkpoint &operator=(checkpoint const &c)
checkpoint &operator=(checkpoint &&c)

const_iterator begin() const
const_iterator end() const
std::size_t size() const

char *data()
char const *data() const
```

Private Functions

```
template<typename Archive>
void serialize(Archive &arch, const unsigned int)
```

Private Members

```
std::vector<char> data_
```

Friends

```
friend hpx::util::hpx::serialization::access
std::ostream &operator<< (std::ostream &ost, checkpoint const &ckp)
    Operator<< Overload
```

This overload is the main way to write data from a checkpoint to an object such as a file. Inside the function, the size of the checkpoint will be written to the stream before the checkpoint's data. The operator>> overload uses this to read the correct number of bytes. Be mindful of this additional write and read when you use different facilities to write out or read in data to a checkpoint!

Parameters

- `ost`: Output stream to write to.
- `ckp`: Checkpoint to copy from.

Return Operator`<<` returns the ostream object.

```
std::istream &operator>> (std::istream &ist, checkpoint &ckp)
    Operator>> Overload
```

This overload is the main way to read in data from an object such as a file to a checkpoint. It is important to note that inside the function, the first variable to be read is the size of the checkpoint. This size variable is written to the stream before the checkpoint's data in the operator`<<` overload. Be mindful of this additional read and write when you use different facilities to read in or write out data from a checkpoint!

Parameters

- `ist`: Input stream to write from.
- `ckp`: Checkpoint to write to.

Return Operator`>>` returns the ostream object.

```
template<typename T, typename ...Ts>
void restore_checkpoint (checkpoint const &c, T &t, Ts&... ts)
    Restore_checkpoint
```

Restore_checkpoint takes a checkpoint object as a first argument and the containers which will be filled from the byte stream (in the same order as they were placed in save_checkpoint). Restore_checkpoint can resurrect a stored component in two ways: by passing in a instance of a component's shared_ptr or by passing in an instance of the component's client.

Return Restore_checkpoint returns void.

Template Parameters

- `T`: A container to restore.
- `Ts`: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- `c`: The checkpoint to restore.
- `t`: A container to restore.
- `ts`: Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

```
bool operator== (checkpoint const &lhs, checkpoint const &rhs)
```

```
bool operator!= (checkpoint const &lhs, checkpoint const &rhs)
```

checkpoint_base

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

Functions

```
template<typename Container, typename ...Ts>
void save_checkpoint_data(Container &data, Ts&&... ts)
    save_checkpoint_data
```

Save_checkpoint_data takes any number of objects which a user may wish to store in the given container.

Template Parameters

- Container: Container used to store the check-pointed data.
- Ts: Types of variables to checkpoint

Parameters

- cont: Container instance used to store the checkpoint data
- ts: Variable instances to be inserted into the checkpoint.

```
template<typename ...Ts>
std::size_t prepare_checkpoint_data(Ts const&... ts)
    prepare_checkpoint_data
```

prepare_checkpoint_data takes any number of objects which a user may wish to store in a subsequent save_checkpoint_data operation. The function will return the number of bytes necessary to store the data that will be produced.

Template Parameters

- Ts: Types of variables to checkpoint

Parameters

- ts: Variable instances to be inserted into the checkpoint.

```
template<typename Container, typename ...Ts>
void restore_checkpoint_data(Container const &cont, Ts&... ts)
    restore_checkpoint_data
```

restore_checkpoint_data takes any number of objects which a user may wish to restore from the given container. The sequence of objects has to correspond to the sequence of objects for the corresponding call to save_checkpoint_data that had used the given container instance.

Template Parameters

- Container: Container used to restore the check-pointed data.
- Ts: Types of variables to restore

Parameters

- cont: Container instance used to restore the checkpoint data
- ts: Variable instances to be restored from the container

collectives

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

namespace collectives

Functions

```
template<typename T>
hpx::future<std::vector<std::decay_t<T>>> all_gather(char const *basename, T &&re-
sult, num_sites_arg num_sites =
num_sites_arg(), this_site_arg this_site =
this_site_arg(), generation_arg genera-
tion = generation_arg(), root_site_arg
root_site = root_site_arg())
```

AllGather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.
It will become ready once the all_gather operation has been completed.

Parameters

- `basename`: The base name identifying the all_gather operation
- `local_result`: The value to transmit to all participating sites from this call site.
- `num_sites`: The number of participating sites (default: all localities).
- `generation`: The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once.
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns. \params `root_site` The site that is responsible for creating the all_gather support object. This value is optional and defaults to '0' (zero).

```
template<typename T>
hpx::future<std::vector<std::decay_t<T>>> all_gather(communicator comm, T &&re-
sult, this_site_arg this_site =
this_site_arg())
```

AllGather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.
It will become ready once the all_gather operation has been completed.

Parameters

- `comm`: A communicator object returned from `create_reducer`
- `local_result`: The value to transmit to all participating sites from this call site.
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns.

namespace hpx

namespace collectives

Functions

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> all_reduce(char const *basename, T &&result, F &&op,
                                              num_sites_arg num_sites = num_sites_arg(),
                                              this_site_arg this_site = this_site_arg(), generation_arg generation = generation_arg(),
                                              root_site_arg root_site = root_site_arg())
```

AllReduce a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.
It will become ready once the all_reduce operation has been completed.

Parameters

- `basename`: The base name identifying the all_reduce operation
- `local_result`: The value to transmit to all participating sites from this call site.
- `op`: Reduction operation to apply to all values supplied from all participating sites
- `num_sites`: The number of participating sites (default: all localities).
- `generation`: The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once.
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns. \params `root_site` The site that is responsible for creating the all_reduce support object. This value is optional and defaults to '0' (zero).

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> all_reduce(communicator comm, T &&result, F &&op,
                                              this_site_arg this_site = this_site_arg())
```

AllReduce a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.
It will become ready once the all_reduce operation has been completed.

Parameters

- `comm`: A communicator object returned from `create_reducer`
- `local_result`: The value to transmit to all participating sites from this call site.
- `op`: Reduction operation to apply to all values supplied from all participating sites
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns.

namespace hpx**namespace collectives**

Functions

```
template<typename T>
hpx::future<std::vector<std::decay_t<T>>> all_to_all(char const *basename, T &&re-
sult, num_sites_arg num_sites =
num_sites_arg(), this_site_arg this_site
= this_site_arg(), generation_arg genera-
tion = generation_arg(), root_site_arg
root_site = root_site_arg())
```

AllToAll a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.

It will become ready once the all_to_all operation has been completed.

Parameters

- basename: The base name identifying the all_to_all operation
- local_result: The value to transmit to all participating sites from this call site.
- num_sites: The number of participating sites (default: all localities).
- generation: The generational counter identifying the sequence number of the all_to_all operation performed on the given base name. This is optional and needs to be supplied only if the all_to_all operation on the given base name has to be performed more than once.
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns. \params root_site The site that is responsible for creating the all_to_all support object. This value is optional and defaults to '0' (zero).

```
template<typename T>
hpx::future<std::vector<std::decay_t<T>>> all_to_all(communicator comm, T &&re-
sult, this_site_arg this_site =
this_site_arg())
```

AllToAll a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.

It will become ready once the all_to_all operation has been completed.

Parameters

- comm: A communicator object returned from *create_reducer*
- local_result: The value to transmit to all participating sites from this call site.
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

namespace hpx

namespace collectives

struct generation_arg

Public Functions

```
constexpr generation_arg (std::size_t generation = std::size_t(-1))  
constexpr generation_arg &operator= (std::size_t generation)  
constexpr operator std::size_t () const
```

Public Members

```
std::size_t generation_  
  
struct num_sites_arg
```

Public Functions

```
constexpr num_sites_arg (std::size_t num_sites = std::size_t(-1))  
constexpr num_sites_arg &operator= (std::size_t num_sites)  
constexpr operator std::size_t () const
```

Public Members

```
std::size_t num_sites_  
  
struct root_site_arg
```

Public Functions

```
constexpr root_site_arg (std::size_t root_site = std::size_t(0))  
constexpr root_site_arg &operator= (std::size_t root_site)  
constexpr operator std::size_t () const
```

Public Members

```
std::size_t root_site_  
  
struct tag_arg
```

Public Functions

```
constexpr tag_arg (std::size_t tag = std::size_t(0))  
constexpr tag_arg &operator= (std::size_t tag)  
constexpr operator std::size_t () const
```

Public Members

```
std::size_t tag_
struct that_site_arg
```

Public Functions

```
constexpr that_site_arg(std::size_t that_site = std::size_t(-1))
constexpr that_site_arg &operator=(std::size_t that_site)
constexpr operator std::size_t() const
```

Public Members

```
std::size_t that_site_
struct this_site_arg
```

Public Functions

```
constexpr this_site_arg(std::size_t this_site = std::size_t(-1))
constexpr this_site_arg &operator=(std::size_t this_site)
constexpr operator std::size_t() const
```

Public Members

```
std::size_t this_site_
namespace hpx
```

```
namespace lcos
```

class barrier

#include <barrier.hpp> The barrier is an implementation performing a barrier over a number of participating threads. The different threads don't have to be on the same locality. This barrier can be invoked in a distributed application.

For a local only barrier

See `hpx::lcos::local::barrier`.

Public Functions

barrier (*std::string const &base_name*)

Creates a barrier, rank is locality id, size is number of localities

A barrier *base_name* is created. It expects that `hpx::get_num_localities()` participate and the local rank is `hpx::get_locality_id()`.

Parameters

- *base_name*: The name of the barrier

barrier (*std::string const &base_name, std::size_t num*)

Creates a barrier with a given size, rank is locality id

A barrier *base_name* is created. It expects that *num* participate and the local rank is `hpx::get_locality_id()`.

Parameters

- *base_name*: The name of the barrier
- *num*: The number of participating threads

barrier (*std::string const &base_name, std::size_t num, std::size_t rank*)

Creates a barrier with a given size and rank

A barrier *base_name* is created. It expects that *num* participate and the local rank is *rank*.

Parameters

- *base_name*: The name of the barrier
- *num*: The number of participating threads
- *rank*: The rank of the calling site for this invocation

barrier (*std::string const &base_name, std::vector<std::size_t> const &ranks, std::size_t rank*)

Creates a barrier with a vector of ranks

A barrier *base_name* is created. It expects that *ranks.size()* and the local rank is *rank* (must be contained in *ranks*).

Parameters

- *base_name*: The name of the barrier
- *ranks*: Gives a list of participating ranks (this could be derived from a list of locality ids)
- *rank*: The rank of the calling site for this invocation

void **wait** ()

Wait until each participant entered the barrier. Must be called by all participants

Return This function returns once all participants have entered the barrier (have called *wait*).

hpx::future<void> **wait** (*hpx::launch::async_policy*)

Wait until each participant entered the barrier. Must be called by all participants

Return a future that becomes ready once all participants have entered the barrier (have called *wait*).

Public Static Functions

static void synchronize()

Perform a global synchronization using the default global barrier. The barrier is created once at startup and can be reused throughout the lifetime of an HPX application.

Note This function currently does not support dynamic connection and disconnection of localities.

namespace hpx

namespace collectives

Functions

```
template<typename T>
hpx::future<void> broadcast_to(char const *basename, T &&local_result, num_sites_arg
                                num_sites = num_sites_arg(), this_site_arg this_site =
                                this_site_arg(), generation_arg generation = genera-
                                tion_arg())
```

Broadcast a value to different call sites

This function sends a set of values to all call sites operating on the given base name.

Return This function returns a future that will become ready once the broadcast operation has been completed.

Parameters

- **basename**: The base name identifying the broadcast operation
- **local_result**: A value to transmit to all participating sites from this call site.
- **num_sites**: The number of participating sites (default: all localities).
- **generation**: The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once.
- **this_site**: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
template<typename T>
hpx::future<void> broadcast_to(communicator comm, T &&local_result, this_site_arg
                                this_site = this_site_arg())
```

Broadcast a value to different call sites

This function sends a set of values to all call sites operating on the given base name.

Return This function returns a future that will become ready once the broadcast operation has been completed.

Parameters

- **comm**: A communicator object returned from *create_reducer*
- **local_result**: A value to transmit to all participating sites from this call site.
- **this_site**: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

template<typename T>

```
hpx::future<T> broadcast_from(char const *basename, this_site_arg this_site =
                                this_site_arg(), generation_arg generation = genera-
                                tion_arg())
```

Receive a value that was broadcast to different call sites

This function sends a set of values to all call sites operating on the given base name.

Return This function returns a future holding the value that was sent to all participating sites. It will become ready once the broadcast operation has been completed.

Parameters

- basename: The base name identifying the broadcast operation
- generation: The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once.
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
template<typename T>
hpx::future<T> broadcast_from(communicator comm, this_site_arg this_site = this_site_arg())
```

Receive a value that was broadcast to different call sites

This function sends a set of values to all call sites operating on the given base name.

Return This function returns a future holding the value that was sent to all participating sites. It will become ready once the broadcast operation has been completed.

Parameters

- comm: A communicator object returned from *create_reducer*
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

namespace hpx

namespace lcos

Functions

```
template<typename Action, typename ArgN, ...> hpx::future<std::vector<decltype(Action)>>
```

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<Action>, std::vector<ArgN>)
```

Perform an asynchronous (fire&forget) distributed broadcast operation.

The function `hpx::lcos::broadcast_apply` performs an asynchronous (fire&forget) distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action).

The given action is invoked asynchronously on all given identifiers, and the arguments `ArgN` are passed along to those invocations.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `argN`: [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

```
template<typename Action, typename ArgN, ...>hpx::future< std::vector<decltype(Action)>> hpx::lcos::broadcast_with_index(std::vector<Action>, std::vector<ArgN>, size_t index)
```

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(std::vector<Action>, std::vector<ArgN>, size_t 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.

```
namespace hpx
```

```
namespace collectives
```

Functions

```
hpx::future<channel_communicator> create_channel_communicator (char      const
                                                               *basename,
                                                               num_sites_arg
                                                               num_sites      =
                                                               num_sites_arg(),
                                                               this_site_arg
                                                               this_site      =
                                                               this_site_arg())
```

Create a new communicator object usable with peer-to-peer channel-based operations

This functions creates a new communicator object that can be called in order to pre-allocate a communicator object usable with multiple invocations of channel-based peer-to-peer operations.

Return This function returns a future to a new communicator object usable with the collective operation.

Parameters

- `basename`: The base name identifying the collective operation
- `num_sites`: The number of participating sites (default: all localities).
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns.

```
channel_communicator create_channel_communicator (hpx::launch::sync_policy,
                                                       char      const      *basename,
                                                       num_sites_arg  num_sites      =
                                                       num_sites_arg(),   this_site_arg
                                                       this_site = this_site_arg())
```

Create a new communicator object usable with peer-to-peer channel-based operations

This functions creates a new communicator object that can be called in order to pre-allocate a communicator object usable with multiple invocations of channel-based peer-to-peer operations.

Return This function returns a new communicator object usable with the collective operation.

Parameters

- `basename`: The base name identifying the collective operation
- `num_sites`: The number of participating sites (default: all localities).
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns.

```
template<typename T>
hpx::future<void> set (channel_communicator comm, that_site_arg site, T &&value, tag_arg tag = 0)
```

Send a value to the given site

This function sends a value to the given site based on the given communicator.

Return This function returns a future<void> that becomes ready once the data transfer operation has finished.

Parameters

- `comm`: The channel communicator object to use for the data transfer
- `site`: The destination site
- `value`: The value to send
- `tag`: The (optional) tag identifying the concrete operation

```
template<typename T>
hpx::future<T> get (channel_communicator comm, that_site_arg site, tag_arg tag = 0)
Send a value to the given site
```

This function receives a value from the given site based on the given communicator.

Return This function returns a future<T> that becomes ready once the data transfer operation has finished. The future will hold the received value.

Parameters

- `comm`: The channel communicator object to use for the data transfer
- `site`: The source site

namespace hpx

namespace lcos

Functions

```
hpx::future<hpx::id_type> create_communication_set (char const *basename, std::size_t
num_sites = std::size_t(-1),
std::size_t this_site = std::size_t(-1),
std::size_t arity = std::size_t(-1))
```

The function *create_communication_set* sets up a (distributed) tree-like communication structure that can be used with any of the collective APIs (such like *all_to_all* and similar).

Return This function returns a future holding an id_type of the communicator object to be used on the current locality.

Parameters

- `basename`: The base name identifying the all_to_all operation
- `num_sites`: The number of participating sites (default: all localities).
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns.
- `arity`: The number of children each of the communication nodes is connected to (default: picked based on num_sites)

namespace hpx

namespace collectives

Functions

```
communicator create_communicator(char const *basename, num_sites_arg num_sites
= num_sites_arg(), this_site_arg this_site = this_site_arg(),
generation_arg generation = generation_arg(), root_site_arg root_site = root_site_arg())
```

Create a new communicator object usable with any collective operation

This function creates a new communicator object that can be called in order to pre-allocate a communicator object usable with multiple invocations of any of the collective operations (such as *all_gather*, *all_reduce*, *all_to_all*, *broadcast*, etc.).

Return This function returns a new communicator object usable with the collective operation.

Parameters

- **basename**: The base name identifying the collective operation
- **num_sites**: The number of participating sites (default: all localities).
- **generation**: The generational counter identifying the sequence number of the collective operation performed on the given base name. This is optional and needs to be supplied only if the collective operation on the given base name has to be performed more than once.
- **this_site**: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns. \params **root_site** The site that is responsible for creating the collective support object. This value is optional and defaults to ‘0’ (zero).

namespace hpx

namespace collectives

Functions

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> exclusive_scan(char const *basename, T &&result, F &&op,
num_sites_arg num_sites = num_sites_arg(),
this_site_arg this_site = this_site_arg(),
generation_arg generation = generation_arg(),
root_site_arg root_site = root_site_arg())
```

Exclusive scan a set of values from different call sites

This function performs an exclusive scan operation on a set of values received from all call sites operating on the given base name.

Note The result returned on the `root_site` is always the same as the result returned on `this_site == 1` and is the same as the value provided by the `thje root_site`.

Return This function returns a future holding a vector with all values send by all participating sites. It will become ready once the `exclusive_scan` operation has been completed.

Parameters

- **basename**: The base name identifying the `exclusive_scan` operation
- **local_result**: The value to transmit to all participating sites from this call site.
- **op**: Reduction operation to apply to all values supplied from all participating sites
- **num_sites**: The number of participating sites (default: all localities).
- **generation**: The generational counter identifying the sequence number of the `exclusive_scan` operation performed on the given base name. This is optional and needs to be sup-

plied only if the exclusive_scan operation on the given base name has to be performed more than once.

- **this_site**: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns. \params root_site The site that is responsible for creating the exclusive_scan support object. This value is optional and defaults to '0' (zero).

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> exclusive_scan(communicator comm, T &&result, F &&op,
                                                 this_site_arg this_site = this_site_arg())
```

Exclusive scan a set of values from different call sites

This function performs an exclusive scan operation on a set of values received from all call sites operating on the given base name.

Note The result returned on the root_site is always the same as the result returned on thus_site == 1 and is the same as the value provided by the thje root_site.

Return This function returns a future holding a vector with all values send by all participating sites. It will become ready once the exclusive_scan operation has been completed.

Parameters

- *comm*: A communicator object returned from *create_reducer*
- *local_result*: The value to transmit to all participating sites from this call site.
- *op*: Reduction operation to apply to all values supplied from all participating sites
- *this_site*: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
namespace hpx
```

```
namespace lcos
```

Functions

```
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::f
```

Perform a distributed folding operation.

The function `hpx::lcos::fold_with_index` performs a distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action).

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note The type of the initial value must be convertible to the result type returned from the invoked action.

Return This function returns a future representing the result of the overall folding operation.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `fold_op`: [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- `init`: [in] The initial value to be used for the folding operation
- `argN`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

```
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...>hpx::f
```

Perform a distributed inverse folding operation.

The function `hpx::lcos::inverse_fold` performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action).

Note The type of the initial value must be convertible to the result type returned from the invoked action.

Return This function returns a future representing the result of the overall folding operation.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `fold_op`: [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- `init`: [in] The initial value to be used for the folding operation
- `argN`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

```
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...>hpx::f
```

Perform a distributed inverse folding operation.

The function `hpx::lcos::inverse_fold_with_index` performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action).

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note The type of the initial value must be convertible to the result type returned from the invoked action.

Return This function returns a future representing the result of the overall folding operation.

Parameters

- `ids`: [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `fold_op`: [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- `init`: [in] The initial value to be used for the folding operation
- `argN`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

```
namespace hpx
```

```
namespace collectives
```

Functions

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_here(char const *basename, T &&result,
                                                    num_sites_arg num_sites =
                                                    num_sites_arg(), this_site_arg this_site =
                                                    this_site_arg(), generation_arg generation =
                                                    generation_arg())
```

Gather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

Parameters

- `basename`: The base name identifying the gather operation
- `result`: The value to transmit to the central gather point from this call site.
- `num_sites`: The number of participating sites (default: all localities).
- `generation`: The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied only if the gather operation on the given base name has to be performed more than once.
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_here(communicator comm, T &&result,
                                                       this_site_arg this_site = this_site_arg())
```

Gather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

Parameters

- `comm`: A communicator object returned from `create_reducer`

- **result**: The value to transmit to the central gather point from this call site.
- **this_site**: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_there(char const *basename, T &&result,
                                                       this_site_arg this_site = this_site_arg(),
                                                       generation_arg generation = generation_arg(),
                                                       root_site_arg root_site =
                                                       root_site_arg())
```

Gather a given value at the given call site

This function transmits the value given by *result* to a central gather site (where the corresponding *gather_here* is executed)

Return This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

Parameters

- **basename**: The base name identifying the gather operation
- **result**: The value to transmit to the central gather point from this call site.
- **generation**: The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied only if the gather operation on the given base name has to be performed more than once.
- **this_site**: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **root_site**: The sequence number of the central gather point (usually the locality id). This value is optional and defaults to 0.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_there(communicator comm, T &&result,
                                                       this_site_arg this_site = this_site_arg())
```

Gather a given value at the given call site

This function transmits the value given by *result* to a central gather site (where the corresponding *gather_here* is executed)

Return This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

Parameters

- **comm**: A communicator object returned from *create_reducer*
- **result**: The value to transmit to the central gather point from this call site.
- **this_site**: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
namespace hpx
```

```
namespace collectives
```

Functions

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> inclusive_scan (char const *basename, T &&result, F &&op,
                                                num_sites_arg num_sites = num_sites_arg(),
                                                this_site_arg this_site = this_site_arg(), generation_arg generation = generation_arg(),
                                                root_site_arg root_site = root_site_arg())
```

Inclusive inclusive_scan a set of values from different call sites

This function performs an inclusive scan operation on a set of values received from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.

It will become ready once the inclusive_scan operation has been completed.

Parameters

- basename: The base name identifying the inclusive_scan operation
- local_result: The value to transmit to all participating sites from this call site.
- op: Reduction operation to apply to all values supplied from all participating sites
- num_sites: The number of participating sites (default: all localities).
- generation: The generational counter identifying the sequence number of the inclusive_scan operation performed on the given base name. This is optional and needs to be supplied only if the inclusive_scan operation on the given base name has to be performed more than once.
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns. \params root_site The site that is responsible for creating the inclusive_scan support object. This value is optional and defaults to '0' (zero).

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> inclusive_scan (communicator comm, T &&result, F &&op,
                                                this_site_arg this_site = this_site_arg())
```

Inclusive inclusive_scan a set of values from different call sites

This function performs an inclusive scan operation on a set of values received from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.

It will become ready once the inclusive_scan operation has been completed.

Parameters

- comm: A communicator object returned from *create_reducer*
- local_result: The value to transmit to all participating sites from this call site.
- op: Reduction operation to apply to all values supplied from all participating sites
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

namespace hpx

namespace lcos

```
class latch : public components::client_base<latch, lcos::server::latch>
```

Public Functions

latch()

latch(std::ptrdiff_t count)

Initialize the latch

Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

latch(naming::id_type const &id)

Extension: Create a client side representation for the existing *server::latch* instance with the given global id *id*.

latch(hpx::future<naming::id_type> &&f)

Extension: Create a client side representation for the existing *server::latch* instance with the given global id *id*.

latch(hpx::shared_future<naming::id_type> const &id)

Extension: Create a client side representation for the existing *server::latch* instance with the given global id *id*.

latch(hpx::shared_future<naming::id_type> &&id)

void **count_down_and_wait()**
Decrements counter_ by 1 . Blocks at the synchronization point until counter_ reaches 0.

Requires: counter_ > 0.

Synchronization: Synchronizes with all calls that block on this latch and with all is_ready calls on this latch that return true.

Exceptions

- Nothing.:

void **count_down(std::ptrdiff_t n)**

Decrements counter_ by n. Does not block.

Requires: counter_ >= n and n >= 0.

Synchronization: Synchronizes with all calls that block on this latch and with all is_ready calls on this latch that return true .

Exceptions

- Nothing.:

bool **is_ready() const**

Returns: counter_ == 0. Does not block.

Exceptions

- Nothing.:

void **wait() const**

If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization point until counter_ reaches 0.

Exceptions

- Nothing.:

Private Types

```
typedef components::client_base<latch, lcos::server::latch> base_type
namespace hpx

namespace collectives
```

Functions

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> reduce_here (char const *basename, T &&result, F &&op,
                                              num_sites_arg num_sites = num_sites_arg(),
                                              this_site_arg this_site = this_site_arg(), generation_arg generation = generation_arg())
```

Reduce a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Return This function returns a future holding a vector with all values send by all participating sites.
It will become ready once the all_reduce operation has been completed.

Parameters

- basename: The base name identifying the all_reduce operation
- local_result: A value to reduce on the central reduction point from this call site.
- op: Reduction operation to apply to all values supplied from all participating sites
- num_sites: The number of participating sites (default: all localities).
- generation: The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once.
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
template<typename T, typename F>
hpx::future<decay_t<T>> reduce_here (communicator comm, T &&local_result, F &&op,
                                             this_site_arg this_site = this_site_arg())
```

Reduce a set of values from different call sites

This function receives a set of values that are the result of applying a given operator on values supplied from all call sites operating on the given base name.

Return This function returns a future holding a value calculated based on the values send by all participating sites. It will become ready once the all_reduce operation has been completed.

Parameters

- comm: A communicator object returned from *create_communicator*
- local_result: A value to reduce on the root_site from this call site.
- op: Reduction operation to apply to all values supplied from all participating sites
- this_site: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
template<typename T, typename F>
hpx::future<void> reduce_there (char const *basename, T &&result, this_site_arg this_site =
                                         this_site_arg(), generation_arg generation = generation_arg(),
                                         root_site_arg root_site = root_site_arg())
```

Reduce a given value at the given call site

This function transmits the value given by *result* to a central reduce site (where the corresponding *reduce_here* is executed)

Return This function returns a future<void>. It will become ready once the reduction operation has been completed.

Parameters

- *basename*: The base name identifying the reduction operation
- *result*: A future referring to the value to transmit to the central reduction point from this call site.
- *generation*: The generational counter identifying the sequence number of the reduction operation performed on the given base name. This is optional and needs to be supplied only if the reduction operation on the given base name has to be performed more than once.
- *this_site*: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- *root_site*: The sequence number of the central reduction point (usually the locality id). This value is optional and defaults to 0.

```
template<typename T>
hpx::future<void> reduce_there(communicator comm, T &&local_result, this_site_arg  
this_site = this_site_arg())
```

Reduce a given value at the given call site

This function transmits the value given by *result* to a central reduce site (where the corresponding *reduce_here* is executed)

Return This function returns a future holding a value calculated based on the values send by all participating sites. It will become ready once the all_reduce operation has been completed.

Parameters

- *comm*: A communicator object returned from *create_communicator*
- *local_result*: A value to reduce on the central reduction point from this call site.
- *this_site*: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
namespace hpx
```

```
namespace lcos
```

Functions

```
template<typename Action, typename ReduceOp, typename ArgN, ...>hpx::future<decaytyp  
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 const reference) which will be forwarded to the action invocation.

```
template<typename Action, typename ReduceOp, typename ArgN, ...> hpx::future<decaytype
```

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 const reference) which will be forwarded to the action invocation.

```
namespace hpx
```

```
namespace collectives
```

Functions

```
template<typename T>
hpx::future<T> scatter_from(char const *basename, this_site_arg this_site = this_site_arg(),
                           generation_arg generation = generation_arg(), root_site_arg
                           root_site = root_site_arg())
```

Scatter (receive) a set of values to different call sites

This function receives an element of a set of values operating on the given base name.

Return This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

Parameters

- `basename`: The base name identifying the scatter operation
- `generation`: The generational counter identifying the sequence number of the scatter operation performed on the given base name. This is optional and needs to be supplied only if the scatter operation on the given base name has to be performed more than once.
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever `hpx::get_locality_id()` returns.
- `root_site`: The sequence number of the central scatter point (usually the locality id). This value is optional and defaults to 0.

```
template<typename T>
```

```
hpx::future<T> scatter_from (communicator comm, this_site_arg this_site = this_site_arg())
```

Scatter (receive) a set of values to different call sites

This function receives an element of a set of values operating on the given base name.

Return This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

Parameters

- *comm*: A communicator object returned from *create_reducer*
- *this_site*: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
template<typename T>
```

```
hpx::future<T> scatter_to (char const *basename, std::vector<T> &&result, num_sites_arg  
                           num_sites = num_sites_arg(), this_site_arg this_site =  
                           this_site_arg(), generation_arg generation = generation_arg())
```

Scatter (send) a part of the value set at the given call site

This function transmits the value given by *result* to a central scatter site (where the corresponding *scatter_from* is executed)

Return This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

Parameters

- *basename*: The base name identifying the scatter operation
- *result*: The value to transmit to the central scatter point from this call site.
- *num_sites*: The number of participating sites (default: all localities).
- *generation*: The generational counter identifying the sequence number of the scatter operation performed on the given base name. This is optional and needs to be supplied only if the scatter operation on the given base name has to be performed more than once.
- *this_site*: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

```
template<typename T>
```

```
hpx::future<T> scatter_to (communicator comm, std::vector<T> &&result, this_site_arg  
                           this_site = this_site_arg())
```

Scatter (send) a part of the value set at the given call site

This function transmits the value given by *result* to a central scatter site (where the corresponding *scatter_from* is executed)

Return This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

Parameters

- *comm*: A communicator object returned from *create_reducer*
- *num_sites*: The number of participating sites (default: all localities).
- *this_site*: The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

namespace hpx

namespace lcos

Functions

```
template<typename F, typename ...Args>
hpx::future<void> define_spmd_block(std::string &&name, std::size_t images_per_locality,
                                         F&&, Args&&... args)

struct spmd_block
    #include <spmd_block.hpp> The class spmd_block defines an interface for launching multiple images
    while giving handles to each image to interact with the remaining images. The define_spmd_block
    function templates create multiple images of a user-defined action and launches them in a possibly
    separate thread. A temporary spmd block object is created and diffused to each image. The constraint
    for the action given to the define_spmd_block function is to accept a spmd_block as first parameter.
```

Public Functions

```
spmd_block()

spmd_block(std::string const &name, std::size_t images_per_locality, std::size_t
               num_images, std::size_t image_id)

std::size_t get_images_per_locality() const

std::size_t get_num_images() const

std::size_t this_image() const

void sync_all() const

hpx::future<void> sync_all(hpx::launch::async_policy const&) const

void sync_images(std::set<std::size_t> const &images) const

void sync_images(std::vector<std::size_t> const &input_images) const

template<typename Iterator>
std::enable_if<traits::is_input_iterator<Iterator>::value>::type sync_images(Iterator begin,
                                Iterator end)
const

template<typename ...I>
std::enable_if<util::all_of<typename std::is_integral<I>::value...>::value>::type sync_images(I...
                                         i)

hpx::future<void> sync_images(hpx::launch::async_policy const&, std::set<std::size_t>
                                         const &images) const

hpx::future<void> sync_images(hpx::launch::async_policy const &policy,
                               std::vector<std::size_t> const &input_images) const

template<typename Iterator>
```

```
std::enable_if<traits::is_input_iterator<Iterator>::value, hpx::future<void>>::type sync_images (hpx::launch::async
    const
    &policy,
    Iterator begin,
    Iterator end)
const
```

```
template<typename ...I>
std::enable_if<util::all_of<ttypename std::is_integral<I>::type...>::value, hpx::future<void>>::type sync_images (
```

Private Types

```
using barrier_type = hpx::lcos::barrier
using table_type = std::map<std::set<std::size_t>, std::shared_ptr<barrier_type>>
```

Private Functions

```
template<typename Archive>
void serialize (Archive&, unsigned)
```

Private Members

```
std::string name_
std::size_t images_per_locality_
std::size_t num_images_
std::size_t image_id_
hpx::util::jenkins_hash hash_
std::shared_ptr<hpx::lcos::barrier> barrier_
table_type barriers_
```

Friends

```
friend hpx::lcos::hpx::serialization::access
```

command_line_handling

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace util
```

```
struct command_line_handling
```

Public Functions

```
command_line_handling(runtime_configuration rtcfg, std::vector<std::string> ini_config,
                      function_nonser<int> hpx::program_options::variables_map
                      &vm
                      >hpx_main_f

int call(hpx::program_options::options_description const &desc_cmdline, int argc, char
        **argv, std::vector<std::shared_ptr<components::component_registry_base>> &com-
        ponent_registries)
```

Public Members

```
hpx::program_options::variables_map vm_
util::runtime_configuration rtcfg_
std::vector<std::string> ini_config_
util::function_nonser<int> (hpx::program_options::variables_map &vm) > hpx_main_f_
std::size_t node_
std::size_t num_threads_
std::size_t num_cores_
std::size_t num_localities_
std::size_t pu_step_
std::size_t pu_offset_
std::string queuing_
std::string affinity_domain_
std::string affinity_bind_
std::size_t numa_sensitive_
bool use_process_mask_
bool cmd_line_parsed_
```

```
bool info_printed_
bool version_printed_
```

Protected Functions

```
void check_affinity_domain() const
void check_affinity_description() const
void check_pu_offset() const
void check_pu_step() const
bool handle_arguments (util::manage_config &cfgmap, hpx::program_options::variables_map &vm, std::vector<std::string> &ini_config, std::size_t &node, bool initial = false)
void enable_logging_settings (hpx::program_options::variables_map &vm, std::vector<std::string> &ini_config)
void store_command_line (int argc, char **argv)
void store_unregistered_options (std::string const &cmd_name, std::vector<std::string> const &unregistered_options)
bool handle_help_options (hpx::program_options::options_description const &help)
void handle_attach_debugger ()
std::vector<std::string> preprocess_config_settings (int argc, char **argv)
namespace hpx
```

```
namespace util
```

Functions

```
int handle_late_commandline_options (util::runtime_configuration &ini, hpx::program_options::options_description const &options, void (*handle_print_bind)() std::size_t , void (*handle_list_parcelports)() = nullptr)
namespace hpx
```

```
namespace util
```

Functions

```
bool parse_commandline (hpx::util::section const &rtcfg, hpx::program_options::options_description
                      const &app_options, std::string const &cmdline,
                      hpx::program_options::variables_map &vm, std::size_t node,
                      int error_mode = return_on_error, hpx::runtime_mode mode =
                      runtime_mode::default_, hpx::program_options::options_description
                      *visible = nullptr, std::vector<std::string> *unregistered_options =
                      nullptr)

bool parse_commandline (hpx::util::section const &rtcfg, hpx::program_options::options_description
                      const &app_options, std::string const
                      &arg0, std::vector<std::string> const &args,
                      hpx::program_options::variables_map &vm, std::size_t node,
                      int error_mode = return_on_error, hpx::runtime_mode mode =
                      runtime_mode::default_, hpx::program_options::options_description
                      *visible = nullptr, std::vector<std::string> *unregistered_options =
                      nullptr)
```

components

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

Functions

```
template<typename Client>
std::vector<Client> find_all_from_basename (std::string base_name, std::size_t num_ids)
```

Return all registered clients from all localities from the given base name.

This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

Return all registered ids from all localities from the given base name.

Return A list of futures representing the ids which were registered using the given base name.

Note The futures embedded in the returned client objects will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Template Parameters

- Client: The client type to return

Parameters

- base_name: [in] The base name for which to retrieve the registered ids.
- num_ids: [in] The number of registered ids to expect.

This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

Return A list of futures representing the ids which were registered using the given base name.

Note The futures will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Parameters

- `base_name`: [in] The base name for which to retrieve the registered ids.
- `num_ids`: [in] The number of registered ids to expect.

```
template<typename Client>
std::vector<Client> find_from_basename (std::string base_name, std::vector<std::size_t> const
&ids)
```

Return registered clients from the given base name and sequence numbers.

This function locates the ids which were registered with the given base name and the given sequence numbers. It returns a list of futures representing those ids.

Return registered ids from the given base name and sequence numbers.

Return A list of futures representing the ids which were registered using the given base name and sequence numbers.

Note The futures embedded in the returned client objects will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Template Parameters

- `Client`: The client type to return

Parameters

- `base_name`: [in] The base name for which to retrieve the registered ids.
- `ids`: [in] The sequence numbers of the registered ids.

This function locates the ids which were registered with the given base name and the given sequence numbers. It returns a list of futures representing those ids.

Return A list of futures representing the ids which were registered using the given base name and sequence numbers.

Note The futures will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Parameters

- `base_name`: [in] The base name for which to retrieve the registered ids.
- `ids`: [in] The sequence numbers of the registered ids.

```
template<typename Client>
Client find_from_basename (std::string base_name, std::size_t sequence_nr)
```

Return registered id from the given base name and sequence number.

This function locates the id which was registered with the given base name and the given sequence number. It returns a future representing those id.

This function locates the id which was registered with the given base name and the given sequence number. It returns a future representing those id.

Return A representing the id which was registered using the given base name and sequence numbers.

Note The future embedded in the returned client object will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Template Parameters

- Client: The client type to return

Parameters

- base_name: [in] The base name for which to retrieve the registered ids.
- sequence_nr: [in] The sequence number of the registered id.

Return A representing the id which was registered using the given base name and sequence numbers.

Note The future will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Parameters

- base_name: [in] The base name for which to retrieve the registered ids.
- sequence_nr: [in] The sequence number of the registered id.

```
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)
```

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.

```
template<typename Client>
Client unregister_with_basename (std::string base_name, std::size_t sequence_nr)
```

Unregister the given id using the given base name.

Unregister the given base name.

The function unregisters the given ids using the provided base name.

The function unregisters the given ids using the provided base name.

Return A future representing the result of the un-registration operation itself.

Template Parameters

- Client: The client type to return

Parameters

- base_name: [in] The base name for which to retrieve the registered ids.
- sequence_nr: [in, optional] The sequential number to use for the un-registration. This number has to be the same as has been used with *register_with_basename* before.

Return A future representing the result of the un-registration operation itself.

Parameters

- base_name: [in] The base name for which to retrieve the registered ids.
- sequence_nr: [in, optional] The sequential number to use for the un-registration. This number has to be the same as has been used with *register_with_basename* before.

namespace hpx

Functions

hpx::future<bool> register_with_basename (std::string base_name, hpx::id_type id, std::size_t sequence_nr = ~static_cast<std::size_t>(0))

Register the given id using the given base name.

The function registers the given ids using the provided base name.

Return A future representing the result of the registration operation itself.

Note The operation will fail if the given sequence number is not unique.

Parameters

- base_name: [in] The base name for which to retrieve the registered ids.
- id: [in] The id to register using the given base name.
- sequence_nr: [in, optional] The sequential number to use for the registration of the id. This number has to be unique system wide for each registration using the same base name. The default is the current locality identifier. Also, the sequence numbers have to be consecutive starting from zero.

hpx::future<bool> register_with_basename (std::string base_name, hpx::future<hpx::id_type> f, std::size_t sequence_nr = ~static_cast<std::size_t>(0))

Register the id wrapped in the given future using the given base name.

The function registers the object the given future refers to using the provided base name.

Return A future representing the result of the registration operation itself.

Note The operation will fail if the given sequence number is not unique.

Parameters

- `base_name`: [in] The base name for which to retrieve the registered ids.
- `f`: [in] The future which should be registered using the given base name.
- `sequence_nr`: [in, optional] The sequential number to use for the registration of the id. This number has to be unique system wide for each registration using the same base name. The default is the current locality identifier. Also, the sequence numbers have to be consecutive starting from zero.

```
namespace hpx
```

```
namespace components
```

```
template<typename Component>
class client : public hpx::components::client_base<client<Component>, Component>
```

Public Functions

```
client()
client(naming::id_type const &id)
client(naming::id_type &&id)
client(future_type const &f)
client(future_type &&f)
client(future<naming::id_type> &&f)
client(future<client> &&c)
client(client const &rhs)
client(client &&rhs)
client &operator=(naming::id_type const &id)
client &operator=(naming::id_type &&id)
client &operator=(future_type const &f)
client &operator=(future_type &&f)
client &operator=(future<naming::id_type> &&f)
client &operator=(client const &rhs)
client &operator=(client &&rhs)
```

Private Types

```
template<>
using base_type = client_base<client<Component>, Component>

template<>
using future_type = typename base_type::future_type

namespace hpx

namespace components
```

Functions

```
template<typename Derived, typename Stub>
bool operator==(client_base<Derived, Stub> const &lhs, client_base<Derived, Stub> const
&rhs)

template<typename Derived, typename Stub>
bool operator<(client_base<Derived, Stub> const &lhs, client_base<Derived, Stub> const
&rhs)

template<typename Derived, typename Stub>
class client_base : public detail::make_stub::type<Stub>
```

Public Types

```
template<>
using stub_argument_type = Stub

template<>
using server_component_type = typename detail::make_stub::server_component_type

template<>
using is_client_tag = void
```

Public Functions

```
client_base()

client_base(id_type const &id)
client_base(id_type &&id)
client_base(hpx::shared_future<id_type> const &f)
client_base(hpx::shared_future<id_type> &&f)
client_base(hpx::future<id_type> &&f)
client_base(client_base const &rhs)
client_base(client_base &&rhs)
client_base(hpx::future<Derived> &&d)
```

```

~client_base()

client_base &operator=(id_type const &id)

client_base &operator=(id_type &&id)

client_base &operator=(hpx::shared_future<id_type> const &f)

client_base &operator=(hpx::shared_future<id_type> &&f)

client_base &operator=(hpx::future<id_type> &&f)

client_base &operator=(client_base const &rhs)

client_base &operator=(client_base &&rhs)

bool valid() const

operator bool() const

void free()

id_type const &get_id() const

naming::gid_type const &get_raw_gid() const

hpx::shared_future<id_type> detach()

hpx::shared_future<id_type> share() const

void reset(id_type const &id)

void reset(id_type &&id)

void reset(shared_future<id_type> &&rhs)

id_type const &get() const

bool is_ready() const

bool has_value() const

bool has_exception() const

void wait() const

std::exception_ptr get_exception_ptr() const

template<typename F>
hpx::traits::future_then_result_t<Derived, F> then(launch l, F &&f)

template<typename F>
hpx::traits::future_then_result_t<Derived, F> then(F &&f)

hpx::future<bool> register_as(std::string symbolic_name, bool manage_lifetime = true)

void connect_to(std::string const &symbolic_name)

std::string const &registered_name() const

```

Protected Types

```
template<>
using stub_type = typename detail::make_stub<Stub>::type
template<>
using shared_state_type = lcos::detail::future_data_base<id_type>
template<>
using future_type = shared_future<id_type>
```

Protected Functions

```
client_base(hpx::intrusive_ptr<shared_state_type> const &state)
client_base(hpx::intrusive_ptr<shared_state_type> &&state)
```

Protected Attributes

```
hpx::intrusive_ptr<shared_state_type> shared_state_
```

Private Static Functions

```
template<typename F>
static hpx::traits::future_then_result<Derived, F>::cont_result on_ready(hpx::shared_future<id_type>
&&fut, F f)

static bool register_as_helper(client_base const &f, std::string symbolic_name,
bool manage_lifetime)

namespace serialization
```

Functions

```
template<typename Archive, typename Derived, typename Stub>
void serialize(Archive &ar, ::hpx::components::client_base<Derived, Stub> &f, unsigned version)

namespace hpx

namespace components

template<typename Executor, typename BaseComponent>
struct executor_component : public BaseComponent
```

Public Functions

```
template<typename ...Arg>
executor_component(executor_type const &exec, Arg&&... arg)
```

Public Static Functions

```
static void execute(hpx::threads::thread_function_type const &f)

template<typename Executor_ = Executor>
static void schedule_thread(hpx::naming::address::address_type
                           lva,                               naming::address::component_type,
                           hpx::threads::thread_init_data           &data,
                           hpx::threads::thread_schedule_state)
```

This is the default hook implementation for schedule_thread which forwards to the executor instance associated with this component.

Protected Attributes

```
executor_type exec_
```

Private Types

```
template<>
using base_type = BaseComponent

template<>
using executor_type = Executor

template<>
using this_component_type = typename base_type::this_component_type

namespace hpx
```

Functions

```
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(compa
                                                               Stu
                                                               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 and error.

Note The component instance the returned pointer refers to can not be migrated as long as there is at least one copy of the returned `shared_ptr` alive.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `p`: [in] The parameter `p` represents a placeholder type to turn make the call synchronous.
- `id`: [in] The global id of the component for which the pointer to the underlying memory should be retrieved.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

Template Parameters

- The only template parameter has to be the type of the server side component.

```
template<typename Derived, typename Stub>
std::shared_ptr<typename components::client_base<Derived, Stub>::server_component_type> get_ptr(launch::sync_policy
p,
com-
po-
nents::client_base<
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 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.
- *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.

```
namespace hpx
```

```
namespace components
```

Functions

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::id_type const
&id)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::id_type &&id)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::future<hpx::id_type>
const &id)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::future<hpx::id_type>
&&id)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::shared_future<hpx::id_type>
const &id)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::shared_future<hpx::id_type>
&&id)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::id_type>
const
&ids)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::id_type>
&&ids)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::future<hpx::id_type>
const
&ids)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::future<hpx::id_type>
&&ids)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::shared_future<hpx::id_type>
const
&ids)
```

```
template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::shared_future<hpx::id_type>
&&ids)
```

components_base

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

namespace agas

Functions

```
bool is_console()

bool register_name(launch::sync_policy, std::string const &name, naming::gid_type const &gid, error_code &ec = throws)
bool register_name(launch::sync_policy, std::string const &name, naming::id_type const &id, error_code &ec = throws)
hpx::future<bool> register_name(std::string const &name, naming::id_type const &id)
naming::id_type unregister_name(launch::sync_policy, std::string const &name, error_code &ec = throws)
hpx::future<naming::id_type> unregister_name(std::string const &name)
naming::id_type resolve_name(launch::sync_policy, std::string const &name, error_code &ec = throws)
hpx::future<naming::id_type> resolve_name(std::string const &name)
hpx::future<std::uint32_t> get_num_localities(naming::component_type type = naming::component_invalid)
std::uint32_t get_num_localities(launch::sync_policy, naming::component_type type, error_code &ec = throws)
std::uint32_t get_num_localities(launch::sync_policy, error_code &ec = throws)
std::string get_component_type_name(naming::component_type type, error_code &ec = throws)
hpx::future<std::vector<std::uint32_t>> get_num_threads()
std::vector<std::uint32_t> get_num_threads(launch::sync_policy, error_code &ec = throws)
hpx::future<std::uint32_t> get_num_overall_threads()
std::uint32_t get_num_overall_threads(launch::sync_policy, error_code &ec = throws)
std::uint32_t get_locality_id(error_code &ec = throws)
hpx::naming::gid_type get_locality()
std::vector<std::uint32_t> get_all_locality_ids(naming::component_type type, error_code &ec = throws)
std::vector<std::uint32_t> get_all_locality_ids(error_code &ec = throws)
bool is_local_address_cached(naming::gid_type const &gid, error_code &ec = throws)
```

```
bool is_local_address_cached(naming::gid_type const &gid, naming::address &addr, error_code &ec = throws)

bool is_local_address_cached(naming::id_type const &id, error_code &ec = throws)

bool is_local_address_cached(naming::id_type const &id, naming::address &addr, error_code &ec = throws)

void update_cache_entry(naming::gid_type const &gid, naming::address const &addr, std::uint64_t count = 0, std::uint64_t offset = 0, error_code &ec = throws)

bool is_local_lva_encoded_address(naming::gid_type const &gid)

bool is_local_lva_encoded_address(naming::id_type const &id)

hpx::future<naming::address> resolve(naming::id_type const &id)

naming::address resolve(launch::sync_policy, naming::id_type const &id, error_code &ec = throws)

bool resolve_local(naming::gid_type const &gid, naming::address &addr, error_code &ec = throws)

bool resolve_cached(naming::gid_type const &gid, naming::address &addr)

hpx::future<bool> bind(naming::gid_type const &gid, naming::address const &addr, std::uint32_t locality_id)

bool bind(launch::sync_policy, naming::gid_type const &gid, naming::address const &addr, std::uint32_t locality_id, error_code &ec = throws)

hpx::future<bool> bind(naming::gid_type const &gid, naming::address const &addr, naming::gid_type const &locality_)

bool bind(launch::sync_policy, naming::gid_type const &gid, naming::address const &addr, naming::gid_type const &locality_, error_code &ec = throws)

hpx::future<naming::address> unbind(naming::gid_type const &gid, std::uint64_t count = 1)

naming::address unbind(launch::sync_policy, naming::gid_type const &gid, std::uint64_t count = 1, error_code &ec = throws)

bool bind_gid_local(naming::gid_type const &gid, naming::address const &addr, error_code &ec = throws)

void unbind_gid_local(naming::gid_type const &gid, error_code &ec = throws)

bool bind_range_local(naming::gid_type const &gid, std::size_t count, naming::address const &addr, std::size_t offset, error_code &ec = throws)

void unbind_range_local(naming::gid_type const &gid, std::size_t count, error_code &ec = throws)

void garbage_collect_non_blocking(error_code &ec = throws)

void garbage_collect(error_code &ec = throws)

void garbage_collect_non_blocking(naming::id_type const &id, error_code &ec = throws)
```

Invoke an asynchronous garbage collection step on the given target locality.

```

void garbage_collect (naming::id_type const &id, error_code &ec = throws)
    Invoke a synchronous garbage collection step on the given target locality.

naming::id_type get_console_locality (error_code &ec = throws)
    Return an id_type referring to the console locality.

naming::gid_type get_next_id (std::size_t count, error_code &ec = throws)

void decref (naming::gid_type const &id, std::int64_t credits, error_code &ec = throws)

hpx::future<std::int64_t> incref (naming::gid_type const &gid, std::int64_t credits, naming::id_type const &keep_alive = naming::invalid_id)

std::int64_t incref (launch::sync_policy, naming::gid_type const &gid, std::int64_t credits = 1,
                    naming::id_type const &keep_alive = naming::invalid_id, error_code &ec = throws)

std::int64_t replenish_credits (naming::gid_type &gid)

hpx::future<naming::id_type> get_colocation_id (naming::id_type const &id)

naming::id_type get_colocation_id (launch::sync_policy, naming::id_type const &id, error_code &ec = throws)

hpx::future<hpx::id_type> on_symbol_namespace_event (std::string const &name, bool
                                                       call_for_past_events)

hpx::future<std::pair<naming::id_type, naming::address>> begin_migration (naming::id_type
                                                               const &id)

bool end_migration (naming::id_type const &id)

hpx::future<void> mark_as_migrated (naming::gid_type const &gid,
                                       util::unique_function_nonser<std::pair<bool,
                                                       hpx::future<void>>)
                                         > && fbool expect_to_be_marked_as_migrating

std::pair<bool, components::pinned_ptr> was_object_migrated (naming::gid_type
                                                               const &gid,
                                                               util::unique_function_nonser<components::pinned_
                                                               > && &&

void unmark_as_migrated (naming::gid_type const &gid)

hpx::future<std::map<std::string, hpx::id_type>> find_symbols (std::string const &pattern =
                                                               "*")

std::map<std::string, hpx::id_type> find_symbols (hpx::launch::sync_policy, std::string const
                                                               &pattern = "*")

naming::component_type register_factory (std::uint32_t prefix, std::string const &name,
                                           error_code &ec = throws)

naming::component_type get_component_id (std::string const &name, error_code &ec =
                                           throws)

void destroy_component (naming::gid_type const &gid, naming::address const &addr)

naming::address_type get_primary_ns_lva ()

naming::address_type get_symbol_ns_lva ()

naming::address_type get_runtime_support_lva ()

```

Defines

```
HPX_DEFINE_COMPONENT_COMMANDLINE_OPTIONS (add_options_function)
HPX_REGISTER_COMMANDLINE_MODULE (add_options_function)
HPX_REGISTER_COMMANDLINE_MODULE_DYNAMIC (add_options_function)
namespace hpx

namespace components

struct component_commandline : public component_commandline_base
    #include <component_commandline.hpp> The component_startup_shutdown provides a minimal im-
    plementation of a component's startup/shutdown function provider.
```

Public Functions

hpx::program_options::options_description add_commandline_options ()
Return any additional command line options valid for this component.

Return The module is expected to fill a options_description object with any additional command line options this component will handle.

Note This function will be executed by the runtime system during system startup.

```
namespace cmdline_options_provider
```

Functions

hpx::program_options::options_description add_commandline_options ()

Defines

```
HPX_DEFINE_COMPONENT_STARTUP_SHUTDOWN (startup_, shutdown_)
HPX_REGISTER_STARTUP_SHUTDOWN_MODULE_ (startup, shutdown)
HPX_REGISTER_STARTUP_SHUTDOWN_MODULE (startup, shutdown)
HPX_REGISTER_STARTUP_SHUTDOWN_MODULE_DYNAMIC (startup, shutdown)
HPX_REGISTER_STARTUP_MODULE (startup)
HPX_REGISTER_STARTUP_MODULE_DYNAMIC (startup)
HPX_REGISTER_SHUTDOWN_MODULE (shutdown)
HPX_REGISTER_SHUTDOWN_MODULE_DYNAMIC (shutdown)
namespace hpx

namespace components
```

```
template<bool(*)(startup_function_type &, bool &) Startup, bool(*) (shutdown_function_type &, bool &) Shutdown>
#include <component_startup_shutdown.hpp>
```

The component_startup_shutdown class provides a minimal implementation of a component's startup/shutdown function provider.

Public Functions

`bool get_startup_function (startup_function_type &startup, bool &pre_startup)`
Return any startup function for this component.

Return Returns *true* if the parameter *startup* has been successfully initialized with the startup function.

Parameters

- *startup*: [in, out] The module is expected to fill this function object with a reference to a startup function. This function will be executed by the runtime system during system startup.

`bool get_shutdown_function (shutdown_function_type &shutdown, bool &pre_shutdown)`
Return any shutdown function for this component.

Return Returns *true* if the parameter *shutdown* has been successfully initialized with the shutdown function.

Parameters

- *shutdown*: [in, out] The module is expected to fill this function object with a reference to a shutdown function. This function will be executed by the runtime system during system shutdown.

Defines

```
HPX_DEFINE_GET_COMPONENT_TYPE (component)
HPX_DEFINE_GET_COMPONENT_TYPE_TEMPLATE (template_, component)
HPX_DEFINE_GET_COMPONENT_TYPE_STATIC (component, type)
HPX_DEFINE_COMPONENT_NAME (...)

HPX_DEFINE_COMPONENT_NAME_ (...)
```

`HPX_DEFINE_COMPONENT_NAME_2 (Component, name)`

`HPX_DEFINE_COMPONENT_NAME_3 (Component, name, base_name)`

`namespace hpx`

```
namespace components
```

Typedefs

```
using component_deleter_type = void (*) (hpx::naming::gid_type const&,
                                         hpx::naming::address const&)
```

Enums

```
enum component_enum_type
```

Values:

```
component_invalid = naming::address::component_invalid
component_runtime_support = 0
component_plain_function = 1
component_base_lco = 2
component_base_lco_with_value_unmanaged = 3
component_base_lco_with_value = 4
component_latch = ((5 << 10) | component_base_lco_with_value)
component_barrier = ((6 << 10) | component_base_lco)
component.promise = ((7 << 10) | component_base_lco_with_value)
component_agas_locality_namespace = 8
component_agas_primary_namespace = 9
component_agas_component_namespace = 10
component_agas_symbol_namespace = 11
component_last
component_first_dynamic = component_last
component_upper_bound = 0xfffffL
```

```
enum factory_state_enum
```

Values:

```
factory_enabled = 0
factory_disabled = 1
factory_check = 2
```

Functions

```
bool &enabled(component_type type)
util::atomic_count &instance_count (component_type type)
component_deleter_type &deleter (component_type type)
bool enumerate_instance_counts (util::unique_function_nonser<bool> component_type
> const &f
```

```
const std::string get_component_type_name (component_type type)
    Return the string representation for a given component type id.

constexpr component_type get_base_type (component_type t)
    The lower short word of the component type is the type of the component exposing the actions.

constexpr component_type get_derived_type (component_type t)
    The upper short word of the component is the actual component type.

constexpr component_type derived_component_type (component_type derived, component_type base)
    A component derived from a base component exposing the actions needs to have a specially formatted component type.

constexpr bool types_are_compatible (component_type lhs, component_type rhs)
    Verify the two given component types are matching (compatible)

template<typename Component, typename Enable = void>
constexpr char const *get_component_name ()

template<typename Component, typename Enable = void>
constexpr const char *get_component_base_name ()

template<typename Component>
component_type get_component_type ()

template<typename Component>
void set_component_type (component_type type)
```

namespace naming

Functions

```
std::ostream &operator<< (std::ostream&, address const&)

namespace hpx

namespace components
```

Typedefs

```
typedef abstract_component_base<Component> instead

template<typename Component, typename Derived>
class managed_component
    #include <managed_component_base.hpp> The managed_component template is used as a indirection layer for components allowing to gracefully handle the access to non-existing components.
```

Additionally it provides memory management capabilities for the wrapping instances, and it integrates the memory management with the AGAS service. Every instance of a *managed_component* gets assigned a global id. The provided memory management allocates the *managed_component* instances from a special heap, ensuring fast allocation and avoids a full network round trip to the AGAS service for each of the allocated instances.

Template Parameters

- Component:
- Derived:

```
namespace hpx
```

```
namespace util
```

class unique_id_ranges

```
#include <generate_unique_ids.hpp>
```

The `unique_id_ranges` class is a type responsible for generating unique ids for components, parcels, threads etc.

Public Functions

```
unique_id_ranges()
```

```
naming::gid_type get_id(std::size_t count = 1)
```

Generate next unique component id.

```
void set_range(naming::gid_type const &lower, naming::gid_type const &upper)
```

Private Types

```
enum [anonymous]
```

size of the id range returned by command_getidrange FIXME: is this a policy?

Values:

```
range_delta = 0x100000
```

```
using mutex_type = hpx::util::spinlock
```

Private Members

```
mutex_type mtx_
```

```
naming::gid_type lower_
```

The range of available ids for components.

```
naming::gid_type upper_
```

```
namespace hpx
```

```
template<typename Component, typename Enable = void>
struct get_lva
```

```
#include <get_lva.hpp>
```

The `get_lva` template is a helper structure allowing to convert a local virtual address as stored in a local address (returned from the function `resolver_client::resolve`) to the address of the component implementing the action.

The default implementation uses the template argument `Component` to deduce the type wrapping the component implementing the action. This is used to get the needed address.

Template Parameters

- Component: This is the type of the component implementing the action to execute.

Public Static Functions

```
static constexpr Component *call (naming::address_type lva)
namespace hpx
```

```
namespace components
```

```
class pinned_ptr
```

Public Functions

```
constexpr pinned_ptr()
~pinned_ptr()
pinned_ptr(pinned_ptr const&)
pinned_ptr(pinned_ptr &&rhs)
pinned_ptr &operator=(pinned_ptr const&)
pinned_ptr &operator=(pinned_ptr &&rhs)
```

Public Static Functions

```
template<typename Component>
static pinned_ptr create (naming::address_type lva)
```

Private Functions

```
template<typename Component>
pinned_ptr (naming::address_type lva, id<Component>)
```

Private Members

```
detail::pinned_ptr_base *data_ = nullptr
namespace hpx
```

```
namespace components
```

```
template<typename ServerComponent>
struct stub_base
```

Public Types

```
template<>
using server_component_type = ServerComponent
```

Public Static Functions

```
static components::component_type get_component_type()

namespace hpx

namespace components

template<typename Component>
class abstract_fixed_component_base : private hpx::traits::detail::fixed_component_tag
```

Public Types

```
template<>
using wrapping_type = fixed_component<Component>

template<>
using this_component_type = Component

template<>
using base_type_holder = Component
```

Public Functions

```
virtual ~abstract_fixed_component_base()

virtual naming::address get_current_address() const
```

Public Static Functions

```
static constexpr component_type get_component_type()

static constexpr void set_component_type(component_type t)

namespace hpx

namespace components

template<typename BaseComponent, typename Mutex = lcos::local::spinlock>
struct abstract_base_migration_support : public BaseComponent
#include <abstract_migration_support.hpp> This hook has to be inserted into the derivation chain of
any abstract_component_base for it to support migration.
```

Public Types

```
template<>
using decorates_action = void
```

Public Functions

```
virtual ~abstract_base_migration_support()

virtual void pin() = 0

virtual bool unpin() = 0

virtual std::uint32_t pin_count() const = 0

virtual void mark_as_migrated() = 0

virtual hpx::future<void> mark_as_migrated(hpx::id_type const &to_migrate) = 0

virtual void on_migrated() = 0
```

Public Static Functions

```
template<typename F>
static threads::thread_function_type decorate_action(naming::address_type lva, F &&f)
```

Protected Functions

```
threads::thread_result_type thread_function(threads::thread_function_type &&f, components::pinned_ptr, threads::thread_restart_state state)
```

Private Types

```
template<>
using base_type = BaseComponent

template<>
using this_component_type = typename base_type::this_component_type

template<typename Derived, typename Base>
struct abstract_migration_support : public hpx::components::migration_support<Derived>, public Base
#include <abstract_migration_support.hpp> This hook has to be inserted into the derivation chain of any component for it to support migration.
```

Public Types

```
template<>
using base_type = migration_support<Derived>

template<>
using abstract_base_type = Base

template<>
using wrapping_type = typename base_type::wrapping_type

template<>
using wrapped_type = typename base_type::wrapped_type

template<>
using type_holder = Derived

template<>
using base_type_holder = Base
```

Public Functions

```
abstract_migration_support()

template<typename T, typename ...Ts, typename Enable = std::enable_if_t<!std::is_same_v<std::decay_t<T>, abstract_migration_support>
abstract_migration_support (T &&t, Ts&&... ts)

~abstract_migration_support()

constexpr void finalize()

hpx::future<void> mark_as_migrated (hpx::id_type const &to_migrate)

void mark_as_migrated ()

std::uint32_t pin_count () const

void pin ()

bool unpin ()

void on_migrated ()

namespace hpx

namespace traits

template<typename Component, typename Enable>
struct component_heap_type
```

Public Types

```
template<>
using type = hpx::components::detail::simple_heap<Component>
```

Defines

```
HPX_REGISTER_COMPONENT_HEAP (Component)
```

```
namespace hpx
```

```
namespace components
```

Functions

```
template<typename Component>
Component::heap_type &component_heap()
```

```
namespace hpx
```

```
namespace components
```

```
namespace server
```

Functions

```
template<typename Component, typename ...Ts>
naming::gid_type create (Ts&&... ts)
```

Create a component and forward the passed parameters.

Create arrays of components using their default constructor.

```
template<typename Component, typename ...Ts>
naming::gid_type create_migrated (naming::gid_type const &gid, void **p, Ts&&... ts)
```

```
template<typename Component, typename ...Ts>
std::vector<naming::gid_type> bulk_create (std::size_t count, Ts&&... ts)
```

Create count components and forward the passed parameters.

```
namespace hpx
```

```
namespace components
```

```
namespace server
```

Functions

```
template<typename Component, typename ...Ts>
naming::gid_type construct (Ts&&... ts)

namespace hpx

namespace components

template<typename BaseComponent, typename Mutex = lcos::local::spinlock>
struct locking_hook : public BaseComponent
#include <locking_hook.hpp> This hook can be inserted into the derivation chain of any component
allowing to automatically lock all action invocations for any instance of the given component.
```

Public Types

```
template<>
using decorates_action = void
```

Public Functions

```
locking_hook ()

template<typename T, typename ...Ts, typename Enable = std::enable_if_t<!std::is_same_v<std::decay_t<T>, locking_hook>>
locking_hook (T &&t, Ts&&... ts)

locking_hook (locking_hook const &rhs)

locking_hook (locking_hook &&rhs)

locking_hook &operator= (locking_hook const &rhs)

locking_hook &operator= (locking_hook &&rhs)
```

Public Static Functions

```
template<typename F>
static threads::thread_function_type decorate_action (naming::address_type lva, F
&&f)
```

Protected Types

```
template<>
using yield_decorator_type = util::function_nonser<threads::thread_arg_type (threads::thread_result_type)
```

Protected Functions

```
threads::thread_result_type thread_function(threads::thread_function_type f,
                                             threads::thread_arg_type state)
threads::thread_arg_type yield_function(threads::thread_result_type state)
```

Private Types

```
template<>
using mutex_type = Mutex

template<>
using base_type = BaseComponent

template<>
using this_component_type = typename base_type::this_component_type
```

Private Members

```
mutex_type mtx_

struct decorate_wrapper
```

Public Functions

```
template<typename F, typename Enable = std::enable_if_t<!std::is_same_v<std::decay_t<F>, decorate_wrapper>>
decorate_wrapper(F &&f)

template<>
~decorate_wrapper()

struct undecorate_wrapper
```

Public Functions

```
template<>
undecorate_wrapper()

template<>
~undecorate_wrapper()
```

Public Members

```
template<>
yield_decorator_type yield_decorator_

template<>
struct init<traits::construct_with_back_ptr>
```

Public Static Functions

```
template<typename Component, typename Managed>
static constexpr void call (Component*, Managed*)

template<typename Component, typename Managed, typename ...Ts>
static void call_new (Component *component, Managed *this_, Ts&&... vs)

template<>
struct init<traits::construct_without_back_ptr>
```

Public Static Functions

```
template<typename Component, typename Managed>
static void call (Component *component, Managed *this_)

template<typename Component, typename Managed, typename ...Ts>
static void call_new (Component *component, Managed *this_, Ts&&... vs)

template<>
struct destroy_backptr<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```
template<typename BackPtr>
static void call (BackPtr *back_ptr)

template<>
struct destroy_backptr<traits::managed_object_controls_lifetime>
```

Public Static Functions

```
template<typename BackPtr>
static constexpr void call (BackPtr*)

template<>
struct manage_lifetime<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```
template<typename Component>
static constexpr void call (Component*)

template<typename Component>
static void addr (Component *component)

template<typename Component>
static void release (Component *component)

template<>
struct manage_lifetime<traits::managed_object_controls_lifetime>
```

Public Static Functions

```
template<typename Component>
static void call (Component *component)

template<typename Component>
static constexpr void addref (Component*)

template<typename Component>
static constexpr void release (Component*)

namespace hpx
```

```
namespace components
```

Functions

```
template<typename Component, typename Derived>
void intrusive_ptr_add_ref (managed_component<Component, Derived> *p)

template<typename Component, typename Derived>
void intrusive_ptr_release (managed_component<Component, Derived> *p)

namespace detail_adl_barrier

template<>
struct destroy_backptr<traits::managed_object_controls_lifetime>
```

Public Static Functions

```
template<typename BackPtr>
static constexpr void call (BackPtr*)

template<>
struct destroy_backptr<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```
template<typename BackPtr>
static void call (BackPtr *back_ptr)

template<>
struct init<traits::construct_with_back_ptr>
```

Public Static Functions

```
template<typename Component, typename Managed>
static constexpr void call (Component*, Managed*)

template<typename Component, typename Managed, typename ...Ts>
static void call_new (Component *&component, Managed *this_, Ts&&... vs)

template<>
struct init<traits::construct_without_back_ptr>
```

Public Static Functions

```
template<typename Component, typename Managed>
static void call (Component *component, Managed *this_)

template<typename Component, typename Managed, typename ...Ts>
static void call_new (Component *&component, Managed *this_, Ts&&... vs)

template<>
struct manage_lifetime<traits::managed_object_controls_lifetime>
```

Public Static Functions

```
template<typename Component>
static void call (Component *component)

template<typename Component>
static constexpr void addr (Component*)

template<typename Component>
static constexpr void release (Component*)

template<>
struct manage_lifetime<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```
template<typename Component>
static constexpr void call (Component*)

template<typename Component>
static void addr (Component *component)

template<typename Component>
static void release (Component *component)
```

```
namespace hpx
```

```
namespace components
```

```
template<typename BaseComponent, typename Mutex = lcos::local::spinlock>
```

```
struct migration_support : public BaseComponent
```

#include <migration_support.hpp> This hook has to be inserted into the derivation chain of any component for it to support migration.

Public Types

```
template<>
using decorates_action = void
```

Public Functions

```
migration_support()
```

```
template<typename T, typename ...Ts, typename Enable = std::enable_if_t<!std::is_same_v<std::decay_t<T>, migration_support>>
```

```
migration_support(T &&t, Ts&&... ts)
```

```
~migration_support()
```

```
naming::gid_type get_base_gid(naming::gid_type const &assign_gid = naming::invalid_gid) const
```

```
void pin()
```

```
bool unpin()
```

```
std::uint32_t pin_count() const
```

```
void mark_as_migrated()
```

```
hpx::future<void> mark_as_migrated(hpx::id_type const &to_migrate)
```

```
constexpr void on_migrated()
```

This hook is invoked on the newly created object after the migration has been finished

Public Static Functions

```
static constexpr bool supports_migration()
```

```
template<typename F>
```

```
static threads::thread_function_type decorate_action(naming::address_type lva, F &&f)
```

```
static std::pair<bool, components::pinned_ptr> was_object_migrated(hpx::naming::gid_type const &id, naming::address_type lva)
```

Protected Functions

```
threads::thread_result_type thread_function(threads::thread_function_type  
    &&f,  
                                         components::pinned_ptr,  
                                         threads::thread_restart_state state)
```

Private Types

```
template<>  
using mutex_type = Mutex  
template<>  
using base_type = BaseComponent  
template<>  
using this_component_type = typename base_type::this_component_type
```

Private Members

```
mutex_type mtx_  
std::uint32_t pin_count_  
hpx::lcos::local::promise<void> trigger_migration_  
bool was_marked_for_migration_  
namespace hpx
```

```
namespace util
```

```
class one_size_heap_list  
Subclassed by hpx::components::detail::wrapper_heap_list< Heap >
```

Public Types

```
using list_type = std::list<std::shared_ptr<util::wrapper_heap_base>>  
using iterator = typename list_type::iterator  
using const_iterator = typename list_type::const_iterator  
using mutex_type = lcos::local::spinlock  
using heap_parameters = wrapper_heap_base::heap_parameters
```

Public Functions

```
one_size_heap_list()

template<typename Heap>
one_size_heap_list(char const *class_name, heap_parameters parameters, Heap* = nullptr)

template<typename Heap>
one_size_heap_list(std::string const &class_name, heap_parameters parameters, Heap* = nullptr)

~one_size_heap_list()

void *alloc(std::size_t count = 1)

bool reschedule(void *p, std::size_t count)

void free(void *p, std::size_t count = 1)

bool did_alloc(void *p) const

std::string name() const
```

Public Members

```
std::shared_ptr<util::wrapper_heap_base> (*create_heap_)(char const*, std::size_t, heap_parameters)

const heap_parameters parameters_
```

Protected Attributes

```
mutex_type mtx_

list_type heap_list_
```

Private Members

```
const std::string class_name_
```

Private Static Functions

```
template<typename Heap>
static std::shared_ptr<util::wrapper_heap_base> create_heap(char const *name, std::size_t counter, heap_parameters parameters)
```

Defines

```
HPX_DEBUG_WRAPPER_HEAP
namespace hpx

namespace util

struct wrapper_heap_base
    Subclassed by hpx::components::detail::wrapper_heap
```

Public Functions

```
virtual ~wrapper_heap_base()

virtual bool alloc(void **result, std::size_t count = 1) = 0

virtual bool did_alloc(void *p) const = 0

virtual void free(void *p, std::size_t count = 1) = 0

virtual naming::gid_type get_gid(util::unique_id_ranges &ids, void *p, components::component_type type) = 0

virtual std::size_t heap_count() const = 0

virtual std::size_t size() const = 0

virtual std::size_t free_size() const = 0

struct heap_parameters
```

Public Members

```
std::size_t capacity

std::size_t element_alignment

std::size_t element_size

namespace hpx

namespace traits
```

Variables

```
template<typename Action>
constexpr bool has_decorates_action_v = has_decorates_action<Action>::value

template<typename Component>
constexpr bool component_decorates_action_v = component_decorates_action<Component>::value

template<typename Action, typename Enable = void>
struct action_decorate_function
```

Public Static Functions

```
template<typename F>
static threads::thread_function_type call (naming::address_type lva, F &&f)
```

Public Static Attributes

```
constexpr bool value = has_decorates_action_v<Action>
```

```
template<typename Component, typename Enable = void>
struct component_decorate_function
```

Public Static Functions

```
template<typename F>
static threads::thread_function_type call (naming::address_type lva, F &&f)
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename Component, typename Enable = void>
struct component_config_data
```

Public Static Functions

```
static char const *call ()
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename Component, typename Enable = void>
struct component_pin_support
```

Public Static Functions

```
static constexpr void pin (Component *p)
```

```
static constexpr bool unpin (Component *p)
```

```
static constexpr std::uint32_t pin_count (Component *p)
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename Component, typename Enable = void>
struct component_supports_migration
```

Public Static Functions

```
static constexpr bool call()  
namespace hpx
```

```
namespace components
```

Typedefs

```
using component_type = std::int32_t  
namespace traits
```

```
template<typename Component, typename Enable = void>  
struct component_type_database
```

Subclassed by hpx::traits::component_type_database< Component const, Enable >

Public Static Functions

```
static components::component_type get()  
static void set(components::component_type)
```

Public Static Attributes

```
components::component_type value = components::component_type(-1)  
namespace hpx
```

```
namespace traits
```

```
template<typename Component, typename Enable = void>  
struct component_type_is_compatible
```

Public Static Functions

```
static bool call(naming::address const &addr)  
namespace hpx
```

```
namespace traits
```

Variables

```
template<typename Component>
constexpr bool is_component_v = is_component<Component>::value

template<typename Component>
constexpr bool is_component_or_component_array_v = is_component_or_component_array<Component>::value

template<typename Component>
constexpr bool is_fixed_component_v = is_fixed_component<Component>::value

template<typename Component>
constexpr bool is_managed_component_v = is_managed_component<Component>::value

template<typename Component>
struct is_fixed_component : public std::is_base_of<traits::detail::fixed_component_tag, Component>
    Subclassed by hpx::traits::is_fixed_component<Component const >

template<typename Component>
struct is_managed_component : public std::is_base_of<traits::detail::managed_component_tag, Component>
    Subclassed by hpx::traits::is_managed_component<Component const >

template<typename Component>
struct managed_component_ctor_policy<Component, util::always_void_t<typename Component::has_managed_compo
```

Public Types

```
template<>
using type = typename Component::ctor_policy

template<typename Component>
struct managed_component_dtor_policy<Component, util::always_void<typename Component::has_managed_compon
```

Public Types

```
template<>
using type = typename Component::dtor_policy

namespace hpx
```

```
namespace traits
```

Typedefs

```
template<typename T>
using managed_component_ctor_policy_t = typename managed_component_ctor_policy<T>::type

template<typename T>
using managed_component_dtor_policy_t = typename managed_component_dtor_policy<T>::type

template<typename T, typename Enable = void>
struct managed_component_ctor_policy
```

Public Types

```
template<>
using type = construct_without_back_ptr
```

```
template<typename Component>
struct managed_component_ctor_policy<Component, util::always_void_t<typename Component::has_managed_ctors>
```

Public Types

```
template<>
using type = typename Component::ctor_policy
```

```
template<typename T, typename Enable = void>
struct managed_component_dtor_policy<Component, util::always_void_t<typename Component::has_managed_dtors>
```

Public Types

```
template<>
using type = managed_object_controls_lifetime
```

```
template<typename Component>
struct managed_component_dtor_policy<Component, util::always_void_t<typename Component::has_managed_dtors>
```

Public Types

```
template<>
using type = typename Component::dtor_policy
```

compute

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

```
namespace compute
```

Functions

```
template<typename T, typename Allocator>
void swap(vector<T, Allocator> &x, vector<T, Allocator> &y)
    Effects: x.swap(y);
```

```
template<typename T, typename Allocator = std::allocator<T>>
class vector
```

Public Types

```

template<>
using value_type = T
    Member types (FIXME: add reference to std.

template<>
using allocator_type = Allocator

template<>
using access_target = typename alloc_traits::access_target

template<>
using size_type = std::size_t

template<>
using difference_type = std::ptrdiff_t

template<>
using reference = typename alloc_traits::reference

template<>
using const_reference = typename alloc_traits::const_reference

template<>
using pointer = typename alloc_traits::pointer

template<>
using const_pointer = typename alloc_traits::const_pointer

template<>
using iterator = detail::iterator<T, Allocator>

template<>
using const_iterator = detail::iterator<T const, Allocator>

template<>
using reverse_iterator = detail::reverse_iterator<T, Allocator>

template<>
using const_reverse_iterator = detail::const_reverse_iterator<T, Allocator>

```

Public Functions

```

vector (Allocator const &alloc = Allocator())
vector (size_type count, T const &value, Allocator const &alloc = Allocator())
vector (size_type count, Allocator const &alloc = Allocator())

template<typename InIter, typename Enable = typename std::enable_if<hpx::traits::is_input_iterator<InIter>::valueEnable>
vector (InIter first, InIter last, Allocator const &alloc)

vector (vector const &other)
vector (vector const &other, Allocator const &alloc)
vector (vector &&other)
vector (vector &&other, Allocator const &alloc)

```

```
vector (std::initializer_list<T> init, Allocator const &alloc)

~vector ()

vector &operator=(vector const &other)

vector &operator=(vector &&other)

allocator_type get_allocator () const
    Returns the allocator associated with the container.

reference operator[] (size_type pos)

const_reference operator[] (size_type pos) const

pointer data ()
    Returns pointer to the underlying array serving as element storage. The pointer is such that range [data(); data() + size()) is always a valid range, even if the container is empty (data() is not dereferenceable in that case).

const_pointer data () const
    Returns pointer to the underlying array serving as element storage. The pointer is such that range [data(); data() + size()) is always a valid range, even if the container is empty (data() is not dereferenceable in that case).

T *device_data () const
    Returns a raw pointer corresponding to the address of the data allocated on the device.

std::size_t size () const

std::size_t capacity () const

bool empty () const
    Returns: size() == 0.

void resize (size_type)
    Effects: If size <= size(), equivalent to calling pop_back() size() - size times. If size() < size, appends size - size() default-inserted elements to the sequence.

    Requires: T shall be MoveInsertable and DefaultInsertable into *this.

    Remarks: If an exception is thrown other than by the move constructor of a non-CopyInsertable T there are no effects.

void resize (size_type, T const&)
    Effects: If size <= size(), equivalent to calling pop_back() size() - size times. If size() < size, appends size - size() copies of val to the sequence.

    Requires: T shall be CopyInsertable into *this.

    Remarks: If an exception is thrown there are no effects.

iterator begin ()

iterator end ()

const_iterator cbegin () const

const_iterator cend () const

const_iterator begin () const
```

```

const_iterator end() const

void swap(vector &other)
    Effects: Exchanges the contents and capacity() of *this with that of x.
    Complexity: Constant time.

void clear()
    Effects: Erases all elements in the range [begin(),end()). Destroys all elements in a. Invalidates all references, pointers, and iterators referring to the elements of a and may invalidate the past-the-end iterator.

    Post: a.empty() returns true.
    Complexity: Linear.

```

Private Types

```
typedef traits::allocator_traits<Allocator> alloc_traits
```

Private Members

```

size_type size_
size_type capacity_
allocator_type alloc_
pointer data_

```

```
namespace hpx
```

```
namespace compute
```

```
namespace host
```

```

template<typename T, typename Executor = hpx::parallel::execution::restricted_thread_pool_executor>
struct block_allocator : public hpx::compute::host::detail::policy_allocator<T, hpx::execution::parallel_p
#include <block_allocator.hpp> The block_allocator allocates blocks of memory evenly divided onto the passed vector of targets. This is done by using first touch memory placement.

```

This allocator can be used to write NUMA aware algorithms:

```

using allocator_type = hpx::compute::host::block_allocator<int>; using vector_type =
hpx::compute::vector<int, allocator_type>;
auto numa_nodes = hpx::compute::host::numa_domains(); std::size_t N = 2048; vector_type v(N,
allocator_type(numa_nodes));

```

Public Types

```
template<>
using executor_type = block_executor<Executor>

template<>
using executor_parameters_type = typename executor_type::executor_parameters_type

template<>
using policy_type = hpx::execution::parallel_policy_shim<executor_type, executor_parameters_type>

template<>
using base_type = detail::policy_allocator<T, policy_type>

template<>
using target_type = std::vector<host::target>
```

Public Functions

```
block_allocator()
block_allocator(target_type const &targets)
block_allocator(target_type &&targets)
target_type const &target() const

template<typename Executor>
struct executor_execution_category<compute::host::block_executor<Executor>>
```

Public Types

```
typedef hpx::execution::parallel_execution_tag type
namespace hpx

namespace compute

namespace host

template<typename Executor = hpx::parallel::execution::restricted_thread_pool_executor>
struct block_executor
#include <block_executor.hpp> The block executor can be used to build NUMA aware programs.
It will distribute work evenly across the passed targets
```

Template Parameters

- Executor: The underlying executor to use

Public Types

```
template<>
using executor_parameters_type = hpx::execution::static_chunk_size
```

Public Functions

```
block_executor(std::vector<host::target> const &targets, threads::thread_priority
priority = threads::thread_priority::high, threads::thread_stacksize
stacksize = threads::thread_stacksize::default_,
threads::thread_schedule_hint schedulehint = {})

block_executor(std::vector<host::target> &&targets)

block_executor(block_executor const &other)

block_executor(block_executor &&other)

block_executor &operator=(block_executor const &other)

block_executor &operator=(block_executor &&other)

template<typename F, typename ...Ts>
void post(F &&f, Ts&&... ts)

template<typename F, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute(F
&&f,
Ts&&...
ts)

template<typename F, typename ...Ts>
hpx::util::detail::invoke_deferred_result<F, Ts...>::type sync_execute(F &&f, Ts&&...
ts)

template<typename F, typename Shape, typename ...Ts>
std::vector<hpx::future<typename parallel::execution::detail::bulk_function_result<F, Shape, Ts...>::type>> bu

template<typename F, typename Shape, typename ...Ts>
parallel::execution::detail::bulk_execute_result<F, Shape, Ts...>::type bulk_sync_execute(F
&&f,
Shape
const
&shape,
Ts&&...
ts)

std::vector<host::target> const &targets() const
```

Private Functions

```
void init_executors()
```

Private Members

```
std::vector<host::target> targets_
std::atomic<std::size_t> current_
std::vector<Executor> executors_
threads::thread_priority priority_ = threads::thread_priority::high
threads::thread_stacksize stacksize_ = threads::thread_stacksize::default_
threads::thread_schedule_hint schedulehint_ = {}
```

```
namespace parallel
```

```
namespace execution
```

```
template<typename Executor>
struct executor_execution_category<compute::host::block_executor<Executor>>
```

Public Types

```
typedef hpx::execution::parallel_execution_tag type
```

```
namespace hpx
```

```
namespace compute
```

```
namespace host
```

Functions

```
std::vector<target> get_local_targets()
```

```
namespace hpx
```

```
namespace parallel
```

```
namespace util
```

```
template<typename T, typename Executors>
class numa_allocator
```

Public Types

```
typedef T value_type
typedef value_type *pointer
typedef value_type const *const_pointer
typedef value_type &reference
typedef value_type const &const_reference
typedef std::size_t size_type
typedef std::ptrdiff_t difference_type
```

Public Functions

```
numa_allocator (Executors const &executors, hpx::threads::topology &topo)
numa_allocator (numa_allocator const &rhs)

template<typename U>
numa_allocator (numa_allocator<U, Executors> const &rhs)

pointer address (reference r)
const_pointer address (const_reference r)

pointer allocate (size_type cnt, const void* = nullptr)
void deallocate (pointer p, size_type cnt)

size_type max_size () const
void construct (pointer p, const T &t)
void destroy (pointer p)
```

Private Types

```
typedef Executors::value_type executor_type
```

Private Members

```
Executors const &executors_
hpx::threads::topology &topo_
```

Friends

```
friend hpx::parallel::util::numa_allocator  
  
bool operator==(numa_allocator const&, numa_allocator const&)  
  
bool operator!=(numa_allocator const &l, numa_allocator const &r)  
  
template<typename U>  
struct rebind
```

Public Types

```
template<>  
typedef numa_allocator<U, Executors> other
```

Defines

```
NUMA_BINDING_ALLOCATOR_DEBUG  
namespace hpx
```

Functions

```
static hpx::debug::enable_print<NUMA_BINDING_ALLOCATOR_DEBUG> hpx::nba_deb("NUM_B_A")  
namespace compute  
  
namespace host
```

Typedefs

```
template<typename T>  
using numa_binding_helper_ptr = std::shared_ptr<numa_binding_helper<T>>  
  
template<typename T>  
struct numa_binding_allocator  
#include <numa_binding_allocator.hpp> The numa_binding_allocator allocates memory using  
a policy based on hwloc flags for memory binding. This allocator can be used to request data that  
is bound to one or more numa domains via the bitmap mask supplied
```

Public Types

```
typedef T value_type  
typedef T *pointer  
typedef T &reference  
typedef T const &const_reference  
typedef std::size_t size_type  
typedef std::ptrdiff_t difference_type
```

```
template<>
using numa_binding_helper_ptr = std::shared_ptr<numa_binding_helper<T>>
```

Public Functions

```
numa_binding_allocator()

numa_binding_allocator(threads::hpx_hwloc_membind_policy policy, unsigned int flags)
numa_binding_allocator(numa_binding_helper_ptr bind_func,
threads::hpx_hwloc_membind_policy policy, unsigned int flags)
numa_binding_allocator(numa_binding_allocator const &rhs)

template<typename U>
numa_binding_allocator(numa_binding_allocator<U> const &rhs)
numa_binding_allocator(numa_binding_allocator &&rhs)
numa_binding_allocator &operator=(numa_binding_allocator const &rhs)
numa_binding_allocator &operator=(numa_binding_allocator &&rhs)
pointer address(reference x) const
const_pointer address(const_reference x) const
pointer allocate(size_type n)
void deallocate(pointer p, size_type n)
size_type max_size() const
template<class U, class ...A>
void construct(U *const p, A&&... args)
template<class U>
void destroy(U *const p)
int get_numa_domain(void *page)
std::string get_page_numa_domains(void *addr, std::size_t len) const
void initialize_pages(pointer p, size_t n) const
std::string display_binding(pointer p, numa_binding_helper_ptr helper)
template<typename Binder>
std::shared_ptr<Binder> get_binding_helper_cast() const
```

Public Members

```
const typedef T* hpx::compute::host::numa_binding_allocator::const_pointer  
std::shared_ptr<numa_binding_helper<T>> binding_helper_  
threads::hpx_hwloc_membind_policy policy_  
unsigned int flags_
```

Protected Functions

```
std::vector<threads::hwloc_bitmap_ptr> create_nodesets(threads::hwloc_bitmap_ptr  
bitmap) const  
void touch_pages(pointer p, size_t n, numa_binding_helper_ptr helper, size_type  
numa_domain, std::vector<threads::hwloc_bitmap_ptr> const  
&nodesets) const  
void bind_pages(pointer p, size_t n, numa_binding_helper_ptr helper, size_type  
numa_domain, std::vector<threads::hwloc_bitmap_ptr> const &node-  
sets) const
```

Private Members

```
std::mutex init_mutex  
template<typename U>  
struct rebind
```

Public Types

```
template<>  
typedef numa_binding_allocator<U> other  
template<typename T>  
struct numa_binding_helper
```

Public Functions

```
virtual std::size_t operator() (const T*const, const T*const, std::size_t  
const, std::size_t const) const  
virtual ~numa_binding_helper()  
virtual const std::string &pool_name() const  
virtual std::size_t memory_bytes() const  
virtual std::size_t array_rank() const  
virtual std::size_t array_size(std::size_t) const  
virtual std::size_t memory_step(std::size_t) const  
virtual std::size_t display_step(std::size_t) const
```

```
virtual std::string description() const
```

Public Members

```
std::string pool_name_ = "default"
```

```
namespace parallel
```

```
namespace execution
```

```
template<>
```

```
struct pool numa_hint<numa_binding_allocator_tag>
```

Public Functions

```
int operator() (int const &domain) const
```

```
namespace hpx
```

```
namespace compute
```

```
namespace host
```

Functions

```
std::vector<target> numa_domains()
```

```
namespace hpx
```

```
namespace compute
```

```
namespace host
```

```
struct target
```

Public Functions

```
target()
```

```
target(hpx::threads::mask_type mask)
```

```
native_handle_type &native_handle()
```

```
native_handle_type const &native_handle() const
```

```
std::pair<std::size_t, std::size_t> num_pus() const
```

```
void synchronize() const
```

hpx::future<void> **get_future () const**

Public Static Functions

static std::vector<target> get_local_targets ()

Private Functions

void serialize (serialization::input_archive &ar, const unsigned int)

void serialize (serialization::output_archive &ar, const unsigned int)

Private Members

native_handle_type **handle_**

Friends

friend hpx::compute::host::hpx::serialization::access

bool operator== (target const &lhs, target const &rhs)

struct native_handle_type

Public Functions

native_handle_type ()

native_handle_type (hpx::threads::mask_type mask)

hpx::threads::mask_type &get_device ()

hpx::threads::mask_type const &get_device () const

Private Members

hpx::threads::mask_type **mask_**

Friends

friend hpx::compute::host::target

template<>

struct access_target<host::target>

Public Types

```
typedef host::target target_type
```

Public Static Functions

```
template<typename T>
static T const &read(target_type const&, T const *t)
```

```
template<typename T>
static void write(target_type const&, T *dst, T const *src)
```

```
template<>
struct access_target<std::vector<host::target>>
```

Public Types

```
typedef std::vector<host::target> target_type
```

Public Static Functions

```
template<typename T>
static T const &read(target_type const&, T const *t)
```

```
template<typename T>
static void write(target_type const&, T *dst, T const *src)
```

```
namespace hpx
```

```
namespace compute
```

```
namespace traits
```

```
template<>
struct access_target<host::target>
```

Public Types

```
typedef host::target target_type
```

Public Static Functions

```
template<typename T>
static T const &read(target_type const&, T const *t)
```

```
template<typename T>
static void write(target_type const&, T *dst, T const *src)
```

```
template<>
struct access_target<std::vector<host::target>>
```

Public Types

```
typedef std::vector<host::target> target_type
```

Public Static Functions

```
template<typename T>
static T const &read(target_type const&, T const *t)
```

```
template<typename T>
static void write(target_type const&, T *dst, T const *src)
```

```
namespace hpx
```

```
namespace serialization
```

Functions

```
template<typename T, typename Allocator>
void serialize(input_archive &ar, compute::vector<T, Allocator> &v, unsigned)
```

```
template<typename T, typename Allocator>
void serialize(output_archive &ar, compute::vector<T, Allocator> const &v, unsigned)
```

```
namespace hpx
```

```
namespace compute
```

```
namespace traits
```

```
template<typename Allocator>
struct allocator_traits
```

Public Types

```
template<>
using reference = typename detail::get_reference_type<Allocator>::type
```

```
template<>
using const_reference = typename detail::get_const_reference_type<Allocator>::type
```

```
template<>
using access_target = typename detail::get_target_traits<Allocator>::type
```

```
template<>
using target_type = typename access_target::target_type
```

Public Static Functions

```
static auto target (Allocator const &alloc)
template<typename ...Ts>
static void bulk_construct (Allocator &alloc, pointer p, size_type count, Ts&&... vs)
static void bulk_destroy (Allocator &alloc, pointer p, size_type count)
```

Private Types

```
template<>
using base_type = std::allocator_traits<Allocator>
```

distribution_policies

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace components
```

Variables

```
constexpr char const *const default_binpacking_counter_name = "/runtime{locality/total}/count/compo-
```

```
const binpacking_distribution_policy binpacked = {}
```

A predefined instance of the binpacking *distribution_policy*. It will represent the local locality and will place all items to create here.

```
struct binpacking_distribution_policy
```

#include <binpacking_distribution_policy.hpp> This class specifies the parameters for a binpacking distribution policy to use for creating a given number of items on a given set of localities. The binpacking policy will distribute the new objects in a way such that each of the localities will equalize the number of overall objects of this type based on a given criteria (by default this criteria is the overall number of objects of this type).

Public Functions

```
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 = 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() (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 = de-
                                         fault_binpacking_counter_name) const
```

Create a new *default_distribution* policy representing the given locality

Parameters

- `loc`: [in] The locality the new instance should represent
- `perf_counter_name`: [in] The name of the performance counter that should be used as the distribution criteria (by default the overall number of existing instances of the given component type will be used).

```
template<typename Component, typename ...Ts>
hpx::future<hpx::id_type> create(Ts&&... vs) const
```

Create one object on one of the localities associated by this policy instance

Return A future holding the global address which represents the newly created object

Parameters

- `vs`: [in] The arguments which will be forwarded to the constructor of the new object.

```
template<typename Component, typename ...Ts>
hpx::future<std::vector<bulk_locality_result>> bulk_create(std::size_t count, Ts&&... vs)
                                         const
```

Create multiple objects on the localities associated by this policy instance

Return A future holding the list of global addresses which represent the newly created objects

Parameters

- `count`: [in] The number of objects to create
- `vs`: [in] The arguments which will be forwarded to the constructors of the new objects.

```
std::string const &get_counter_name() const
```

Returns the name of the performance counter associated with this policy instance.

```
std::size_t get_num_localities() const
```

Returns the number of associated localities for this distribution policy

Note This function is part of the creation policy implemented by this class

namespace hpx

namespace components**Variables****const colocating_distribution_policy colocated = {}**

A predefined instance of the co-locating *distribution_policy*. It will represent the local locality and will place all items to create here.

struct colocating_distribution_policy

#include <colocating_distribution_policy.hpp> This class specifies the parameters for a distribution policy to use for creating a given number of items on the locality where a given object is currently placed.

Public Functions**colocating_distribution_policy()**

Default-construct a new instance of a colocating_distribution_policy. This policy will represent the local locality.

colocating_distribution_policy operator() (id_type const &id) const

Create a new colocating_distribution_policy representing the locality where the given object is current located

Parameters

- *id*: [in] The global address of the object with which the new instances should be 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 current located

Parameters

- *client*: [in] The client side representation of the object with which the new instances should be colocated on

*template<typename Component, typename ...Ts>**hpx::future<hpx::id_type> create (Ts&&... vs) const*

Create one object on the locality of the object this distribution policy instance is associated with

Note This function is part of the placement policy implemented by this class

Return A future holding the global address which represents the newly created object

Parameters

- *vs*: [in] The arguments which will be forwarded to the constructor of the new object.

*template<typename Component, typename ...Ts>**hpx::future<std::vector<bulk_locality_result>> bulk_create (std::size_t count, Ts&&... vs)***const**

Create multiple objects colocated with the object represented by this policy instance

Note This function is part of the placement policy implemented by this class

Return A future holding the list of global addresses which represent the newly created objects
Parameters

- `count`: [in] The number of objects to create
- `vs`: [in] The arguments which will be forwarded to the constructors of the new objects.

```
template<typename Action, typename ...Ts>
async_result<Action>::type async (launch policy, Ts&&... vs) const

template<typename Action, typename Callback, typename ...Ts>
async_result<Action>::type async_cb (launch policy, Callback &&cb, Ts&&... vs) const
```

Note This function is part of the invocation policy implemented by this class

```
template<typename Action, typename Continuation, typename ...Ts>
bool apply (Continuation &&c, threads::thread_priority priority, Ts&&... vs) const
```

Note This function is part of the invocation policy implemented by this class

```
template<typename Action, typename ...Ts>
bool apply (threads::thread_priority priority, Ts&&... vs) const
```

```
template<typename Action, typename Continuation, typename Callback, typename ...Ts>
bool apply_cb (Continuation &&c, threads::thread_priority priority, Callback &&cb, Ts&&...
vs) const
```

Note This function is part of the invocation policy implemented by this class

```
template<typename Action, typename Callback, typename ...Ts>
bool apply_cb (threads::thread_priority priority, Callback &&cb, Ts&&... vs) const
```

```
std::size_t get_num_localities () const
```

Returns the number of associated localities for this distribution policy

Note This function is part of the creation policy implemented by this class

```
hpx::id_type get_next_target () const
```

Returns the locality which is anticipated to be used for the next async operation

```
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<Ac
namespace hpx
```

Variables

```
const container_distribution_policy container_layout = {}

struct container_distribution_policy : public default_distribution_policy
```

Public Functions

```
container_distribution_policy()

container_distribution_policy operator() (std::size_t num_partitions) const

container_distribution_policy operator() (hpx::id_type const &locality) const

container_distribution_policy operator() (std::vector<id_type> const &localities) const

container_distribution_policy operator() (std::vector<id_type> &&localities) const

container_distribution_policy operator() (std::size_t num_partitions, std::vector<id_type>
                                         const &localities) const

container_distribution_policy operator() (std::size_t num_partitions, std::vector<id_type>
                                         &&localities) const

std::size_t get_num_partitions() const

std::vector<hpx::id_type> get_localities() const
```

Private Functions

```
template<typename Archive>
void serialize(Archive &ar, const unsigned int)

container_distribution_policy (std::size_t num_partitions, std::vector<id_type> const
                             &localities)

container_distribution_policy (std::size_t num_partitions, std::vector<id_type> &&lo-
                             calities)

container_distribution_policy (hpx::id_type const &locality)
```

Private Members

```
std::size_t num_partitions_
```

Friends

```
friend hpx::hpx::serialization::access

namespace traits

template<>
struct num_container_partitions<container_distribution_policy>
```

Public Static Functions

```
static std::size_t call (container_distribution_policy const &policy)  
namespace hpx  
  
namespace components
```

Variables

const target_distribution_policy **target** = {}

A predefined instance of the target_distribution_policy. It will represent the local locality and will place all items to create here.

struct target_distribution_policy

#include <target_distribution_policy.hpp> This class specifies the parameters for a simple distribution policy to use for creating (and evenly distributing) a given number of items on a given set of localities.

Public Functions

target_distribution_policy()

Default-construct a new instance of a target_distribution_policy. This policy will represent one locality (the local locality).

target_distribution_policy operator() (id_type const &id) const

Create a new target_distribution_policy representing the given locality

Parameters

- loc: [in] The locality the new instance should represent

template<typename Component, typename ...Ts>

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

```

```

template<typename Actionstruct async_result

```

```

#include <target_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<Ac
namespace hpx

```

```

namespace components

```

```

struct unwrapping_result_policy

```

```

#include <unwrapping_result_policy.hpp> This class is a distribution policy that can be using with
actions that return futures. For those actions it is possible to apply certain optimizations if the action
is invoked synchronously.

```

Public Functions

```
unwrapping_result_policy(id_type const &id)

template<typename Client, typename Stub>
unwrapping_result_policy(client_base<Client, Stub> const &client)

template<typename Action, typename ...Ts>
async_result<Action>::type async (launch policy, Ts&&... vs) const

template<typename Action, typename ...Ts>
async_result<Action>::type async (launch::sync_policy, Ts&&... vs) const

template<typename Action, typename Callback, typename ...Ts>
async_result<Action>::type async_cb (launch policy, Callback &&cb, Ts&&... vs) const

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

hpx::id_type const &get_next_target() const

template<typename Action>
struct async_result
```

Public Types

```
template<>
using type = typename traits::promise_local_result::type
```

executors_distributed

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace parallel
```

```
namespace execution
```

Functions

```
template<typename DistPolicy>
distribution_policy_executor<typename std::decay<DistPolicy>::type> make_distribution_policy_executor
```

Create a new *distribution_policy_executor* from the given distribution_policy.

Parameters

- *policy*: The distribution_policy to create an executor from

```
template<typename DistPolicy>
class distribution_policy_executor
#include <distribution_policy_executor.hpp> A distribution_policy_executor creates groups of parallel execution agents which execute in threads implicitly created by the executor and placed on any of the associated localities.
```

Template Parameters

- *DistPolicy*: The distribution policy type for which an executor should be created. The expression *hpx::traits::is_distribution_policy<DistPolicy>::value* must evaluate to true.

Public Functions

```
template<typename DistPolicy_, typename Enable = typename std::enable_if<!std::is_same<distribution_
distribution_policy_executor(DistPolicy_ &&policy)
```

Create a new distribution_policy executor from the given distribution policy

Parameters

- *policy*: The distribution_policy to create an executor from

Private Members

DistPolicy **policy_**

include

See *Public API* for a list of names and headers that are part of the public *HPX API*.

init_runtime

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace hpx

Functions

```
int finalize(double shutdown_timeout, double localwait = -1.0, error_code &ec = throws)
```

Main function to gracefully terminate the HPX runtime system.

The function *hpx::finalize* is the main way to (gracefully) exit any HPX application. It should be called from one locality only (usually the console) and it will notify all connected localities to finish execution. Only after all other localities have exited this function will return, allowing to exit the console locality as well.

During the execution of this function the runtime system will invoke all registered shutdown functions (see *hpx::init*) on all localities.

The default value (-1.0) will try to find a globally set timeout value (can be set as the configuration parameter *hpx.shutdown_timeout*), and if that is not set or -1.0 as well, it will disable any timeout, each connected locality will wait for all existing HPX-threads to terminate.

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*).

namespace hpx

Functions

```
int init (std::function<int> hpx::program_options::variables_map&
          > f, int argc, char **argv, init_params const &params = init_params() Main entry point for launching
          the HPX runtime system.
```

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users *main* () function. It will set up the HPX runtime environment and schedule the function given by *f* as a HPX thread. This overload will not call *hpx_main*.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users *main* () function. It will set up the HPX runtime environment and schedule the function given by *f* as a HPX thread.

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 `parametermode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the `hpx::init` function (See documentation of `hpx::init_params`)

```
int init (std::function<int> int, char**>  
        >f, int argc, char **argv, init_params const &params = init_params())Main entry point for launching  
the HPX runtime system.
```

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

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 `parametermode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the `hpx::init` function (See documentation of `hpx::init_params`)

```
int init (int argc, char **argv, init_params const &params = init_params())  
Main entry point for launching the HPX runtime system.
```

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

Return The function returns the value, which has been returned from the user supplied `f`.

Note If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc/argv`. Otherwise it will be executed as specified by the `parametermode`.

Parameters

- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the `hpx::init` function (See documentation of `hpx::init_params`)

```
int init (std::nullptr_t f, int argc, char **argv, init_params const &params = init_params())
```

Main entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

Return The function returns the value, which has been returned from the user supplied `f`.

Note If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc/argv`. Otherwise it will be executed as specified by the `parametermode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the `hpx::init` function (See documentation of `hpx::init_params`)

```
int init (init_params const &params = init_params())
```

Main entry point for launching the HPX runtime system.

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc/argv`. If no command line arguments are passed, console mode is assumed.

Note If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section ‘[HPX Command Line Options](#)’.

Parameters

- `params`: [in] The parameters to the `hpx::init` function (See documentation of `hpx::init_params`)

```
namespace hpx
```

```
struct init_params
```

```
#include <hpx_init_params.hpp> Parameters used to initialize the HPX runtime through hpx::init and hpx::start.
```

Public Members

```
std::reference_wrapper<hpx::program_options::options_description const> desc_cmdline = detail::default_desc
std::vector<std::string> cfg
startup_function_type startup
shutdown_function_type shutdown
hpx::runtime_mode mode = ::hpx::runtime_mode::default_
hpx::resource::partitioner_mode rp_mode = ::hpx::resource::mode_default_
hpx::resource::rp_callback_type rp_callback
```

```
namespace hpx
```

Functions

```
bool start (std::function<int> hpx::program_options::variables_map&
> f, int argc, char **argv, init_params const &params = init_params()Main non-blocking entry point
for launching the HPX runtime system.
```

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system’s execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system’s execution.

Return The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

Note If the parameter mode is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in argc/argv. Otherwise it will be executed as specified by the parametermode.

Parameters

- f: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If f is nullptr the HPX runtime environment will be started without invoking f.
- argc: [in] The number of command line arguments passed in argv. This is usually the unchanged value as passed by the operating system (to main()).
- argv: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to main()).
- params: [in] The parameters to the `hpx::start` function (See documentation of `hpx::init_params`)

```
bool start (std::function<int> int, char**>f, int argc, char **argv, init_params const &params = init_params())  
Main non-blocking entry point for launching the HPX runtime system.
```

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by f as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by f as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Return The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

Note If the parameter mode is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in argc/argv. Otherwise it will be executed as specified by the parametermode.

Parameters

- f: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If f is nullptr the HPX runtime environment will be started without invoking f.
- argc: [in] The number of command line arguments passed in argv. This is usually the unchanged value as passed by the operating system (to main()).
- argv: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to main()).
- params: [in] The parameters to the `hpx::start` function (See documentation of `hpx::init_params`)

```
bool start (int argc, char **argv, init_params const &params = init_params())  
Main non-blocking entry point for launching the HPX runtime system.
```

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by f as a HPX thread. It will return immediately after that. Use

`hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Return The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

Note If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc/argv`. Otherwise it will be executed as specified by the `parametermode`.

Parameters

- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the `hpx::start` function (See documentation of `hpx::init_params`)

bool **start** (`std::nullptr_t f, int argc, char **argv, init_params const ¶ms = init_params()`)
Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Return The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

Note If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc/argv`. Otherwise it will be executed as specified by the `parametermode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the `hpx::start` function (See documentation of `hpx::init_params`)

```
bool start (init_params const &params = init_params())  
Main non-blocking entry point for launching the HPX runtime system.
```

This is a simplified main, non-blocking entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings). It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Return The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

Note The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc/argv`. If no command line arguments are passed, console mode is assumed.

Note If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section ‘HPX Command Line Options’.

Parameters

- `params`: [in] The parameters to the `hpx::start` function (See documentation of `hpx::init_params`)

namespace hpx

Functions

```
int suspend (error_code &ec = throws)  
Suspend the runtime system.
```

The function `hpx::suspend` is used to suspend the HPX runtime system. It can only be used when running HPX on a single locality. It will block waiting for all thread pools to be empty. This function only be called when the runtime is running, or already suspended in which case this function will do nothing.

Return This function will always return zero.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
int resume (error_code &ec = throws)  
Resume the HPX runtime system.
```

The function `hpx::resume` is used to resume the HPX runtime system. It can only be used when running HPX on a single locality. It will block waiting for all thread pools to be resumed. This function only be called when the runtime suspended, or already running in which case this function will do nothing.

Return This function will always return zero.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Parameters

- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
namespace hpx_startup
```

Functions

```
std::vector<std::string> user_main_config (std::vector<std::string> const &cfg)
```

Variables

```
std::vector<std::string> (*user_main_config_function) (std::vector<std::string> const &)
```

lcos_distributed

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
namespace hpx
```

```
namespace lcos
```

```
template<typename T>
class channel
```

Public Types

```
template<>
using value_type = T
```

Public Functions

```
channel()
```

```
channel (naming::id_type const &loc)
```

```
channel (hpx::future<naming::id_type> &&id)
```

```
channel (hpx::shared_future<naming::id_type> &&id)
```

```
channel (hpx::shared_future<naming::id_type> const &id)
```

```
hpx::future<T> get (launch::async_policy, std::size_t generation = default_generation) const
```

```
hpx::future<T> get (std::size_t generation = default_generation) const
```

```
T get (launch::sync_policy, std::size_t generation = default_generation, hpx::error_code &ec =
       hpx::throws) const
```

```
T get (launch::sync_policy, hpx::error_code &ec, std::size_t generation = default_generation)
      const

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value, bool> set (launch::apply_policy, U val, std::size_t generation = default_generation)

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value, hpx::future<void>> set (launch::async_policy, U val, std::size_t generation = default_generation)

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value> set (launch::sync_policy, U val, std::size_t generation = default_generation)

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value && !traits::is_launch_policy<U>::value> set (U val, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value, bool> set (launch::apply_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value, hpx::future<void>> set (launch::async_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value> set (launch::sync_policy, std::size_t generation = default_generation)

void close (launch::apply_policy, bool force_delete_entries = false)
hpx::future<std::size_t> close (launch::async_policy, bool force_delete_entries = false)
std::size_t close (launch::sync_policy, bool force_delete_entries = false)
std::size_t close (bool force_delete_entries = false)

channel_iterator<T, channel<T>> begin () const
channel_iterator<T, channel<T>> end () const
channel_iterator<T, channel<T>> rbegin () const
channel_iterator<T, channel<T>> rend () const
```

Private Types

```
template<>
using base_type = components::client_base<channel<T>, lcos::server::channel<T>>
```

Private Static Attributes

```
constexpr std::size_t default_generation = std::size_t(-1)
```

```
template<typename T, typename Channel>
class channel_iterator : public hpx::util::iterator_facade<channel_iterator<T, Channel>, T const, std::input_iterator_tag>
```

Public Functions

```
channel_iterator()
```

```
channel_iterator(Channel const &c)
```

Private Types

```
template<>
using base_type = hpx::util::iterator_facade<channel_iterator<T, Channel>, T const, std::input_iterator_tag>
```

Private Functions

```
std::pair<T, bool> get_checked() const
```

```
bool equal(channel_iterator const &rhs) const
```

```
void increment()
```

```
base_type::reference dereference() const
```

Private Members

```
Channel const *channel_
```

```
std::pair<T, bool> data_
```

Friends

```
friend hpx::lcos::hpx::util::iterator_core_access
```

```
template<typename Channel>
class channel_iterator<void, Channel> : public hpx::util::iterator_facade<channel_iterator<void, Channel>, void, std::input_iterator_tag>
```

Public Functions

```
channel_iterator()  
channel_iterator(Channel const &c)
```

Private Types

```
template<>  
using base_type = hpx::util::iterator_facade<channel_iterator<void, Channel>, util::unused_type const, std::in
```

Private Functions

```
bool get_checked()  
bool equal(channel_iterator const &rhs) const  
void increment()  
base_type::reference dereference() const
```

Private Members

```
Channel const *channel_  
bool data_
```

Friends

```
friend hpx::lcos::hpx::util::iterator_core_access  
template<typename T>  
class receive_channel
```

Public Types

```
template<>  
using value_type = T
```

Public Functions

```
receive_channel()  
receive_channel(channel<T> const &c)  
receive_channel(hpx::future<naming::id_type> &&id)  
receive_channel(hpx::shared_future<naming::id_type> &&id)  
receive_channel(hpx::shared_future<naming::id_type> const &id)  
hpx::future<T> get(launch::async_policy, std::size_t generation = default_generation) const
```

```

    hpx::future<T> get (std::size_t generation = default_generation) const
    T get (launch::sync_policy, std::size_t generation = default_generation, hpx::error_code &ec =
            hpx::throws) const
    T get (launch::sync_policy, hpx::error_code &ec, std::size_t generation = default_generation)
            const
    channel_iterator<T, channel<T>> begin () const
    channel_iterator<T, channel<T>> end () const
    channel_iterator<T, channel<T>> rbegin () const
    channel_iterator<T, channel<T>> rend () const

```

Private Types

```

template<>
using base_type = components::client_base<receive_channel<T>, lcos::server::channel<T>>

```

Private Static Attributes

```
constexpr std::size_t default_generation = std::size_t(-1)
```

```
template<typename T>
class send_channel
```

Public Types

```

template<>
using value_type = T

```

Public Functions

```

send_channel ()
send_channel (channel<T> const &c)
send_channel (hpx::future<naming::id_type> &&id)
send_channel (hpx::shared_future<naming::id_type> &&id)
send_channel (hpx::shared_future<naming::id_type> const &id)
template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value, bool> set (launch::apply_policy, U val, std::size_t
generation = default_generation)
template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value, hpx::future<void>> set (launch::async_policy, U
val, std::size_t generation = default_generation)
template<typename U, typename U2 = T>

```

```
std::enable_if_t<!std::is_void<U>::value> set (launch::sync_policy, U val, std::size_t generation = default_generation)

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U>::value && !traits::is_launch_policy<U>::value> set (U
                                         val,
                                         std::size_t
                                         genera-
                                         tion
                                         =
                                         default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value, bool> set (launch::apply_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value, hpx::future<void>> set (launch::async_policy,
                                                               std::size_t generation =
                                                               default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value> set (launch::sync_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value> set (std::size_t generation = default_generation)

void close (launch::apply_policy, bool force_delete_entries = false)

hpx::future<std::size_t> close (launch::async_policy, bool force_delete_entries = false)

std::size_t close (launch::sync_policy, bool force_delete_entries = false)

std::size_t close (bool force_delete_entries = false)
```

Private Types

```
template<>
using base_type = components::client_base<send_channel<T>, lcos::server::channel<T>>
```

Private Static Attributes

```
constexpr std::size_t default_generation = std::size_t(-1)

namespace hpx
```

```
namespace lcos
```

```
template<typename ValueType>
struct object_semaphore : public components::client_base<object_semaphore<ValueType>, lcos::server::object
```

Public Types

```
template<>
using server_type = lcos::server::object_semaphore<ValueType>

template<>
using base_type = components::client_base<object_semaphore, lcos::server::object_semaphore<ValueType>>
```

Public Functions

```
object_semaphore()

object_semaphore (naming::id_type gid)

hpx::future<void> signal (launch::async_policy, ValueType const &val, std::uint64_t count = 1)

void signal (launch::sync_policy, ValueType const &val, std::uint64_t count = 1)

hpx::future<ValueType> get (launch::async_policy)

ValueType get (launch::sync_policy)

future<void> abort_pending (launch::async_policy, error ec = no_success)

void abort_pending (launch::sync_policy, error = no_success)

void wait (launch::async_policy)

void wait (launch::sync_policy)
```

Defines

```
HPX_REGISTER_CHANNEL_DECLARATION (...)

HPX_REGISTER_CHANNEL_DECLARATION_ (...)

HPX_REGISTER_CHANNEL_DECLARATION_1 (type)

HPX_REGISTER_CHANNEL_DECLARATION_2 (type, name)

HPX_REGISTER_CHANNEL (...)

HPX_REGISTER_CHANNEL_ (...)

HPX_REGISTER_CHANNEL_1 (type)

HPX_REGISTER_CHANNEL_2 (type, name)

namespace hpx

namespace lcos

namespace server

template<typename T, typename RemoteType>
class channel
```

Public Types

```
template<>
using base_type_holder = lcos::base_lco_with_value<T, RemoteType, traits::detail::component_tag>

template<>
using wrapping_type = typename base_type::wrapping_type
```

Public Functions

```
channel()

naming::address get_current_address() const

void set_value(RemoteType &&result)

void set_exception(std::exception_ptr const&)

result_type get_value()

result_type get_value(error_code &ec)

hpx::future<T> get_generation(std::size_t generation)

void set_generation(RemoteType &&value, std::size_t generation)

std::size_t close(bool force_delete_entries)
```

Public Static Functions

```
static components::component_type get_component_type()

static void set_component_type(components::component_type type)
```

Private Types

```
template<>
using base_type = components::component_base<channel>

template<>
using result_type = std::conditional_t<std::is_void<T>::value, util::unused_type, T>
```

Private Members

```
lcos::local::channel<result_type> channel_

namespace hpx

namespace lcos

namespace server
```

```
template<typename ValueType>
struct object_semaphore : public components::managed_component_base<object_semaphore<ValueType>
```

Public Types

```
template<>
using base_type = components::managed_component_base<object_semaphore>

template<>
using mutex_type = hpx::lcos::local::spinlock

template<>
using slist_option_type = boost::intrusive::member_hook<queue_thread_entry, typename queue_thre

template<>
using thread_queue_type = boost::intrusive::slist<queue_thread_entry, slist_option_type, boost::intrusive

template<>
using value_slist_option_type = boost::intrusive::member_hook<queue_value_entry, typename que

template<>
using value_queue_type = boost::intrusive::slist<queue_value_entry, value_slist_option_type, boost::intrusive
```

Public Functions

```
object_semaphore()

~object_semaphore()

void signal (ValueType const &val, std::uint64_t count)

void get (naming::id_type const &lco)

void abort_pending (error ec)

void wait()
```

Public Members

```
HPX_DEFINE_COMPONENT_ACTION ( object_semaphore, abort_pending, abort_pending_  
mutex_type mtx_
```

Private Functions

```
void resume (std::unique_lock<mutex_type> &l)  
  
struct queue_thread_entry
```

Public Types

```
template<>
typedef boost::intrusive::slist_member_hook<boost::intrusive::link_mode<boost::intrusive::normal_link>>
```

Public Functions

```
template<>
queue_thread_entry(naming::id_type const &id)
```

Public Members

```
template<>
naming::id_type id_
template<>
hook_type slist_hook_
struct queue_value_entry
```

Public Types

```
template<>
typedef boost::intrusive::slist_member_hook<boost::intrusive::link_mode<boost::intrusive::normal_link>>
```

Public Functions

```
template<>
queue_value_entry(ValueType const &val, std::uint64_t count)
```

Public Members

```
template<>
ValueType val_
template<>
std::uint64_t count_
template<>
hook_type slist_hook_
```

naming

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace naming
```

Functions

```
void decrement_refcnt (gid_type const &gid)
void save (serialization::output_archive &ar, id_type const&, unsigned int)
void load (serialization::input_archive &ar, id_type&, unsigned int)
```

naming_base

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_ADDRESS_VERSION
namespace hpx
```

```
namespace naming
```

Functions

```
std::ostream &operator<< (std::ostream &os, address const &addr)
struct address
```

Public Types

```
using component_type = naming::component_type
using address_type = naming::address_type
```

Public Functions

```
constexpr address ()
constexpr address (gid_type const &l, component_type t = component_invalid)
constexpr address (gid_type const &l, component_type t, void *lva)
constexpr address (void *lva, component_type t = component_invalid)
constexpr address (address_type a)
constexpr operator bool () const
```

Public Members

```
gid_type locality_
component_type type_ = component_invalid
address_type address_ = nullptr
```

Public Static Attributes

```
constexpr const component_type component_invalid = -1
```

Private Functions

```
template<typename Archive>
void save(Archive &ar, unsigned int version) const

template<typename Archive>
void load(Archive &ar, unsigned int version)

HPX_SERIALIZATION_SPLIT_MEMBER()
```

Friends

```
friend hpx::naming::hpx::serialization::access
friend constexpr bool operator==(address const &lhs, address const &rhs)
```

Defines

```
HPX_GIDTYPE_VERSION
template<>
struct hash<hpx::naming::gid_type>
```

Public Functions

```
std::size_t operator() (::hpx::naming::gid_type const &gid) const
namespace hpx

namespace naming
```

Functions

```

gid_type operator+(gid_type const &lhs, gid_type const &rhs)
```

```

gid_type operator-(gid_type const &lhs, gid_type const &rhs)
```

```

void save(serialization::output_archive &ar, gid_type const&, unsigned int)
```

```

void load(serialization::input_archive &ar, gid_type&, unsigned int version)
```

```

gid_type get_gid_from_locality_id(std::uint32_t locality_id)
```

```

std::uint32_t get_locality_id_from_gid(std::uint64_t msb)
```

```

std::uint32_t get_locality_id_from_gid(gid_type const &id)
```

```

gid_type get_locality_from_gid(gid_type const &id)
```

```

bool is_locality(gid_type const &gid)
```

```

std::uint64_t replace_locality_id(std::uint64_t msb, std::uint32_t locality_id)
```

```

gid_type replace_locality_id(gid_type const &gid, std::uint32_t locality_id)
```

```

constexpr bool refers_to_virtual_memory(std::uint64_t msb)
```

```

constexpr bool refers_to_virtual_memory(gid_type const &gid)
```

```

constexpr bool refers_to_local_lva(gid_type const &gid)
```

```

gid_type replace_component_type(gid_type const &gid, std::uint32_t type)
```

```

std::ostream &operator<<(std::ostream &os, gid_type const &id)
```

Variables

```
constexpr const gid_type invalid_gid = {}
```

```
struct gid_type
```

#include <gid_type.hpp> Global identifier for components across the HPX system.

Subclassed by hpx::naming::detail::id_type_impl

Public Types

```

using size_type = gid_type
using difference_type = gid_type
using mutex_type = gid_type
```

Public Functions

```
constexpr gid_type()

constexpr gid_type(std::uint64_t lsb_id)

gid_type(void *lsb_id)

gid_type(std::uint64_t msb_id, std::uint64_t lsb_id)

gid_type(std::uint64_t msb_id, void *lsb_id)

constexpr gid_type(gid_type const &rhs)

constexpr gid_type(gid_type &&rhs)

~gid_type()

gid_type &operator=(std::uint64_t lsb_id)

gid_type &operator=(gid_type const &rhs)

gid_type &operator=(gid_type &&rhs)

constexpr operator bool() const

gid_type &operator++()

gid_type operator++(int)

gid_type &operator--()

gid_type operator--(int)

gid_type operator+=(gid_type const &rhs)

gid_type operator+=(std::uint64_t rhs)

gid_type operator-=(gid_type const &rhs)

gid_type operator-=(std::uint64_t rhs)

constexpr std::uint64_t get_msb() const

constexpr void set_msb(std::uint64_t msb)

constexpr std::uint64_t get_lsb() const

constexpr void set_lsb(std::uint64_t lsb)

void set_lsb(void *lsb)

std::string to_string() const

void lock()

bool try_lock()

void unlock()

mutex_type &get_mutex() const
```

Public Static Attributes

```
constexpr std::uint64_t credit_base_mask = 0x1full
constexpr std::uint16_t credit_shift = 24
constexpr std::uint64_t credit_mask = credit_base_mask << credit_shift
constexpr std::uint64_t was_split_mask = 0x80000000ull
constexpr std::uint64_t has_credits_mask = 0x40000000ull
constexpr std::uint64_t is_locked_mask = 0x20000000ull
constexpr std::uint64_t locality_id_mask = 0xffffffff00000000ull
constexpr std::uint16_t locality_id_shift = 32
constexpr std::uint64_t virtual_memory_mask = 0x3fffffull
constexpr std::uint64_t dont_cache_mask = 0x800000ull
constexpr std::uint64_t is_migratable = 0x400000ull
constexpr std::uint64_t dynamically_assigned = 0x1ull
constexpr std::uint64_t component_type_base_mask = 0xfffffull
constexpr std::uint64_t component_type_shift = 1ull
constexpr std::uint64_t component_type_mask = component_type_base_mask << component_type_shift
constexpr std::uint64_t credit_bits_mask = credit_mask | was_split_mask | has_credits_mask
constexpr std::uint64_t internal_bits_mask = credit_bits_mask | is_locked_mask | dont_cache_mask | is_mig
constexpr std::uint64_t special_bits_mask = locality_id_mask | internal_bits_mask | component_type_mask
```

Private Types

```
using spinlock_pool = util::spinlock_pool<gid_type>
```

Private Functions

```
bool acquire_lock ()
void relinquish_lock ()
constexpr bool is_locked() const
```

Private Members

```
std::uint64_t id_msb_ = 0
std::uint64_t id_lsb_ = 0
```

Friends

```
gid_type operator+ (gid_type const &lhs, gid_type const &rhs)
gid_type operator+ (gid_type const &lhs, std::uint64_t rhs)
gid_type operator- (gid_type const &lhs, gid_type const &rhs)
gid_type operator- (gid_type const &lhs, std::uint64_t rhs)
gid_type operator& (gid_type const &lhs, std::uint64_t rhs)
bool operator== (gid_type const &lhs, gid_type const &rhs)
bool operator!= (gid_type const &lhs, gid_type const &rhs)
bool operator< (gid_type const &lhs, gid_type const &rhs)
bool operator>= (gid_type const &lhs, gid_type const &rhs)
bool operator<= (gid_type const &lhs, gid_type const &rhs)
bool operator> (gid_type const &lhs, gid_type const &rhs)
std::ostream &operator<< (std::ostream &os, gid_type const &id)
void save (serialization::output_archive &ar, gid_type const&, unsigned int)
void load (serialization::input_archive &ar, gid_type&, unsigned int version)
namespace std
```



```
template<>
struct hash<hpx::naming::gid_type>
```

Public Functions

```
std::size_t operator() (::hpx::naming::gid_type const &gid) const
template<>
struct get_remote_result<naming::id_type, naming::gid_type>
```

Public Static Functions

```
static naming::id_type call (naming::gid_type const &rhs)
template<>
struct promise_local_result<naming::gid_type>
```

Public Types

```
typedef naming::id_type type
template<>
struct get_remote_result<std::vector<naming::id_type>, std::vector<naming::gid_type>>
```

Public Static Functions

```
static std::vector<naming::id_type> call (std::vector<naming::gid_type> const &rhs)
template<>
struct promise_local_result<std::vector<naming::gid_type>>
```

Public Types

```
typedef std::vector<naming::id_type> type
namespace hpx
```

```
namespace naming
```

Functions

```
std::ostream &operator<< (std::ostream &os, id_type const &id)
char const *get_management_type_name (id_type::management_type m)
id_type get_id_from_locality_id (std::uint32_t locality_id)
std::uint32_t get_locality_id_from_id (id_type const &id)
id_type get_locality_from_id (id_type const &id)
bool is_locality (id_type const &id)
bool operator!= (id_type const &lhs, id_type const &rhs)
bool operator<= (id_type const &lhs, id_type const &rhs)
bool operator> (id_type const &lhs, id_type const &rhs)
bool operator>= (id_type const &lhs, id_type const &rhs)
```

Variables

```
const id_type invalid_id = id_type()
struct id_type
```

Public Types

```
enum management_type
    Values:
        unknown_deleter = -1
        unmanaged = 0
            unmanaged GID
        managed = 1
            managed GID
        managed_move_credit = 2
            managed GID which will give up all credits when sent
```

Public Functions

```
constexpr id_type()

id_type (std::uint64_t lsb_id, management_type t)

id_type (gid_type const &gid, management_type t)

id_type (std::uint64_t msb_id, std::uint64_t lsb_id, management_type t)

id_type (id_type const &o)

id_type (id_type &&o)

id_type &operator= (id_type const &o)

id_type &operator= (id_type &&o)

gid_type const &get_gid()

gid_type const &get_gid() const

id_type::management_type get_management_type() const

id_type &operator++()

id_type operator++ (int)

operator bool() const

std::uint64_t get_msb() const

void set_msb (std::uint64_t msb)

std::uint64_t get_lsb() const

void set_lsb (std::uint64_t lsb)

void set_lsb (void *lsb)

void make_unmanaged() const

hpx::intrusive_ptr<detail::id_type_impl> &impl()

hpx::intrusive_ptr<detail::id_type_impl> const &impl() const
```

Private Members

```
hpx::intrusive_ptr<detail::id_type_impl> gid_
```

Friends

```
bool operator==(id_type const &lhs, id_type const &rhs)
bool operator!=(id_type const &lhs, id_type const &rhs)
bool operator<(id_type const &lhs, id_type const &rhs)
bool operator<=(id_type const &lhs, id_type const &rhs)
bool operator>(id_type const &lhs, id_type const &rhs)
bool operator>=(id_type const &lhs, id_type const &rhs)

std::ostream &operator<< (std::ostream &os, id_type const &id)
```

namespace traits

```
template<>
struct get_remote_result<naming::id_type, naming::gid_type>
```

Public Static Functions

```
static naming::id_type call (naming::gid_type const &rhs)

template<>
struct get_remote_result<std::vector<naming::id_type>, std::vector<naming::gid_type>>
```

Public Static Functions

```
static std::vector<naming::id_type> call (std::vector<naming::gid_type> const &rhs)

template<>
struct promise_local_result<naming::gid_type>
```

Public Types

```
typedef naming::id_type type

template<>
struct promise_local_result<std::vector<naming::gid_type>>
```

Public Types

```
typedef std::vector<naming::id_type> type  
namespace hpx  
  
namespace naming
```

Typedefs

```
using component_type = std::int32_t  
using address_type = void*
```

Variables

```
constexpr std::uint32_t invalid_locality_id = ~static_cast<std::uint32_t>(0)  
constexpr std::int32_t component_invalid = -1  
namespace hpx  
  
namespace naming
```

Functions

id_type unmanaged(id_type const &id)

The helper function `hpx::unmanaged` can be used to generate a global identifier which does not participate in the automatic garbage collection.

Return This function returns a new global id referencing the same object as the parameter *id*. The only difference is that the returned global identifier does not participate in the automatic garbage collection.

Note This function allows to apply certain optimizations to the process of memory management in HPX. It however requires the user to take full responsibility for keeping the referenced objects alive long enough.

Parameters

- *id*: [in] The id to generated the unmanaged global id from This parameter can be itself a managed or a unmanaged global id.

parcelport_libfabric

See *Public API* for a list of names and headers that are part of the public *HPX* API.

```
template<>  
struct connection_handler_traits<policies::libfabric::parcelport>
```

Public Types

```
template<>
using connection_type = policies::libfabric::sender

template<>
using send_early_parcel = HPX_PARCELPORT_LIBFABRIC_HAVE_BOOTSTRAPPING

template<>
using do_background_work = std::true_type

template<>
using send_immediate_parcels = std::true_type
```

Public Static Functions

```
static constexpr const char *type()

static constexpr const char *pool_name()

static constexpr const char *pool_name_postfix()

namespace hpx

namespace parcelset

template<>
struct connection_handler_traits<policies::libfabric::parcelport>
```

Public Types

```
template<>
using connection_type = policies::libfabric::sender

template<>
using send_early_parcel = HPX_PARCELPORT_LIBFABRIC_HAVE_BOOTSTRAPPING

template<>
using do_background_work = std::true_type

template<>
using send_immediate_parcels = std::true_type
```

Public Static Functions

```
static constexpr const char *type()

static constexpr const char *pool_name()

static constexpr const char *pool_name_postfix()

namespace hpx
```

```
namespace parcelset

namespace policies

namespace libfabric

class fabric_error : public runtime_error
```

Public Functions

```
fabric_error(int err, const std::string &msg)
fabric_error(int err)
int error_code() const
```

Public Members

```
int error_
```

Public Static Functions

```
static char *error_string(int err)
```

```
namespace hpx
```

```
namespace parcelset

namespace policies

namespace libfabric
```

```
template<int SIZE>
struct header
```

Public Types

```
typedef serialization::serialization_chunk chunktype
```

Public Functions

```
template<typename Buffer>
header (Buffer const &buffer, void *tag)

char *chunk_ptr ()

detail::chunk_header *chunk_header_ptr ()

detail::rma_info *rma_info_ptr ()

detail::message_info *message_info_ptr ()

detail::message_chunk *message_chunk_ptr ()

char *message_ptr ()

uint32_t chunk_data_offset () const

uint32_t rma_info_offset () const

uint32_t message_info_offset () const

uint32_t message_offset () const

char *chunk_data ()

char *message_data ()

bool message_piggy_back ()

uint64_t tag ()

uint32_t message_size ()

std::uint32_t header_length ()

void set_message_rdma_info (uint64_t key, const void *addr)

std::uint32_t num_chunks ()

std::uint32_t num_zero_copy_chunks ()

std::uint32_t num_index_chunks ()
```

Public Static Attributes

```
constexpr unsigned int header_block_size = sizeof(detail::header_block)

constexpr unsigned int data_size_ = SIZE - header_block_size

const unsigned int chunk_flag = 0x01

const unsigned int message_flag = 0x02

const unsigned int normal_flag = 0x04

const unsigned int zerocopy_flag = 0x08
```

Private Members

```
detail::header_block message_header
std::array<char, data_size_data_
```

Friends

```
std::ostream &operator<< (std::ostream &os, header<SIZE> &h)
```

Defines

```
HPX_PARCELPORT_LIBFABRIC_ENDPOINT_MSG
namespace hpx
```

```
namespace parcelset
```

```
namespace policies
```

```
namespace libfabric
```

TypeDefs

```
typedef std::shared_ptr<libfabric_controller> libfabric_controller_ptr
class libfabric_controller
```

Public Types

```
typedef hpx::lcos::local::spinlock mutex_type
typedef hpx::parcelset::policies::libfabric::unique_lock<mutex_type> unique_lock
typedef hpx::parcelset::policies::libfabric::scoped_lock<mutex_type> scoped_lock
typedef std::tuple<hpx::lcos::local::promise<fid_ep*>, hpx::shared_future<fid_ep*>> promise_tuple_t
typedef hpx::concurrent::unordered_map<uint32_t, promise_tuple_type>::map_read_lock_type map_read
typedef hpx::concurrent::unordered_map<uint32_t, promise_tuple_type>::map_write_lock_type map_write
typedef util::function_nonser<void (fid_ep *endpoint, uint32_t ipaddr) >
ConnectionFunction
typedef util::function_nonser<void (fid_ep *endpoint, uint32_t ipaddr) >
DisconnectionFunction
```

Public Functions

```

libfabric_controller(std::string const &provider, std::string const &domain,
                      std::string const &endpoint, int = 7910)

void boot_PMI()

~libfabric_controller()

void open_fabric(std::string const &provider, std::string const &domain,
                  std::string const &endpoint_type)

void startup(parcelport *pp)

void _set_check_domain_op_value(int op, const char *value)

void _set_disable_registration()

void create_event_queue()

locality create_local_endpoint()

void new_endpoint_active(struct fi_info *info, struct fid_ep
                         **new_endpoint)

void bind_endpoint_to_queues(struct fid_ep *endpoint)

void bind_passive_endpoint_to_queues(struct fid_pep *endpoint)

void initialize_localities()

fi_addr_t get_fabric_address(const locality &dest_fabric)

const locality &here() const

const bool &immediate_data_supported() const

bool isTerminated()

void setConnectionFunction(ConnectionFunction f)

void setDisconnectionFunction(DisconnectionFunction f)

int poll_endpoints(bool stopped = false)

int poll_for_work_completions()

int poll_send_queue()

int poll_recv_queue()

int poll_event_queue(bool = false)

struct fid_domain *get_domain()

rma_memory_pool<libfabric_region_provider> &get_memory_pool()

void create_completion_queues(struct fi_info *info, int N)

std::pair<bool, hpx::shared_future<struct fid_ep*>> insert_new_future(uint32_t
                           re-
                           mote_ip)

```

```
fi_addr_t insert_address (const locality &remote)

hpx::shared_future<struct fid_ep*> connect_to_server (const locality &remote)

void disconnect_all()

bool active()
```

Public Members

```
hpx::concurrent::unordered_map<uint32_t, promise_tuple_type> endpoint_tmp_
std::unordered_map<uint64_t, fi_addr_t> endpoint_av_
locality here_
locality agas_
struct fi_info *fabric_info_
struct fid_fabric *fabric_
struct fid_domain *fabric_domain_
struct fid_pep *ep_passive_
struct fid_ep *ep_active_
struct fid_ep *ep_shared_rx_ctxt_
struct fid_eq *event_queue_
struct fid_cq *txcq_
struct fid_cq *rxcq_
struct fid_av *av_
bool immediate_
```

Private Types

```
typedef std::chrono::time_point<std::chrono::system_clock> time_type
```

Private Members

```
std::string device_
std::string interface_
sockaddr_in local_addr_
ConnectionFunction connection_function_
DisconnectionFunction disconnection_function_
std::unique_ptr<rma_memory_pool<libfabric_region_provider>> memory_pool_
std::vector<receiver> receivers_
mutex_type initialization_mutex_
mutex_type endpoint_map_mutex_
```

```
mutex_type polling_mutex_
time_type event_check_time_
uint32_t event_pause_

namespace hpx

namespace parcelset

namespace policies

namespace libfabric

struct libfabric_region_provider
```

Public Types

```
typedef struct fid_mr provider_region
typedef struct fid_domain provider_domain
```

Public Static Functions

```
template<typename ...Args>
static int register_memory(Args&&... args)

static int unregister_memory(provider_region *region)

static int flags()
```

```
namespace hpx
```

```
namespace parcelset
```

```
namespace policies
```

```
namespace libfabric
```

```
struct locality
```

Public Types

```
typedef std::array<uint32_t, array_length> locality_data
```

Public Functions

```
locality (const locality_data &in_data)

locality ()

locality (const locality &other)

locality (locality &&other)

locality &operator= (const locality &other)

const uint32_t &ip_address () const

fi_addr_t fi_address () const

void set_fi_address (fi_addr_t fi_addr)

uint16_t port () const

operator bool () const

void save (serialization::output_archive &ar) const

void load (serialization::input_archive &ar)

const void *fabric_data () const

char *fabric_data_writable ()

bool valid ()
```

Public Static Functions

```
static const char *type ()

static const uint32_t &ip_address (const locality_data &data)
```

Public Static Attributes

```
const uint32_t array_length = HPX_PARCELPORT_LIBFABRIC_LOCALITY_SIZE / 4

const uint32_t array_size = HPX_PARCELPORT_LIBFABRIC_LOCALITY_SIZE
```

Private Members

```
locality_data data_
fi_addr_t fi_address_
```

Friends

```
bool operator== (locality const &lhs, locality const &rhs)
bool operator< (locality const &lhs, locality const &rhs)
std::ostream &operator<< (std::ostream &os, locality const &loc)
```

Defines

```
HPX_PARCELPORT_LIBFABRIC_SUSPEND_WAKE
HPX_PARCELPORT_LIBFABRIC_USE_SMALL_VECTOR
HPX_PARCELPORT_LIBFABRIC_IMM_UNSUPPORTED
template<>
struct plugin_config_data<hpx::parcelset::policies::libfabric::parcelport>
```

Public Static Functions

```
static char const *priority()
static void init (int*, char***, util::command_line_handling&)
static void destroy()
static char const *call()

namespace hpx

namespace parcelset

namespace policies

namespace libfabric

struct parcelport : public parcelport_impl<parcelport>
```

Public Types

```
typedef hpx::lcos::local::spinlock mutex_type
typedef hpx::parcelset::policies::libfabric::scoped_lock<mutex_type> scoped_lock
typedef hpx::parcelset::policies::libfabric::unique_lock<mutex_type> unique_lock
typedef rma_memory_region<libfabric_region_provider> region_type
typedef memory_region_allocator<libfabric_region_provider> allocator_type
typedef header<HPX_PARCELPORT_LIBFABRIC_MESSAGE_HEADER_SIZE> header_type
typedef rma_memory_pool<libfabric_region_provider> memory_pool_type
typedef pinned_memory_vector<char, header_size, region_type, memory_pool_type> snd_data_type
typedef parcel_buffer<snd_data_type> snd_buffer_type
```

Public Functions

```
parcelport (util::runtime_configuration const &ini,
            threads::policies::callback_notifier const &notifier)

bool do_run ()

sender *get_connection (parcelset::locality const &dest, fi_addr_t &fi_addr)

void reclaim_connection (sender *s)

std::shared_ptr<sender> create_connection (parcelset::locality const &dest, error_code &ec)

~parcelport ()

bool can_bootstrap () const
    Should not be used any more as parcelport_impl handles this?

std::string get_locality_name () const
    Return the name of this locality.

parcelset::locality agas_locality (util::runtime_configuration const &ini) const

parcelset::locality create_locality () const

void do_stop ()

bool can_send_immediate ()

template<typename Handler>
bool async_write (Handler &&handler, sender *sender, fi_addr_t addr,
                  snd_buffer_type &buffer)

bool background_work (std::size_t num_thread, parcelport_background_mode mode)

void io_service_work ()

bool background_work_OS_thread()
```

Public Members

```
libfabric_controller_ptr libfabric_controller_
uint32_t ip_addr_
bool bootstrap_enabled_
bool parcelport_enabled_
mutex_type stop_mutex
boost::lockfree::stack<sender*, boost::lockfree::capacity<HPX_PARCELPORT_LIBFABRIC_THROTTLE_SE
std::atomic<bool> stopped_
memory_pool_type *chunk_pool_
performance_counter<unsigned int> completions_handled_
performance_counter<unsigned int> senders_in_use_
```

Public Static Functions

```
static void suspended_task_debug (const std::string &match)
```

Public Static Attributes

```
constexpr unsigned int header_size = header_type::header_block_size
```

Private Types

```
typedef parcelport_impl<parcelport> base_type
namespace traits

template<>
struct plugin_config_data<hpx::parcelset::policies::libfabric::parcelport>
```

Public Static Functions

```
static char const *priority()
static void init (int*, char***, util::command_line_handling&)
static void destroy ()
static char const *call ()
```

Defines

```
sched_getcpu()
nhex(n)
hexpointer(p)
hexuint64(p)
hexuint32(p)
hexlength(p)
hexnumber(p)
hexbyte(p)
decimal(n)
decnumber(p)
dec4(p)
ipaddress(p)
sockaddress(p)
THREAD_ID
CRC32(buf, len)
CRC32_MEM(buf, len, txt)
LOG_TRACE_MSG(x)
LOG_DEBUG_MSG(x)
LOG_INFO_MSG(x)
LOG_WARN_MSG(x)
LOG_ERROR_MSG(x)
LOG_FATAL_MSG(x)
LOG_EXCLUSIVE(x)
FUNC_START_DEBUG_MSG
FUNC_END_DEBUG_MSG
LOG_FORMAT_MSG(x)
LOG_DEVEL_MSG(x)
LOG_TIMED_INIT(name)
LOG_TIMED_MSG(name, level, delay, x)
LOG_TIMED_BLOCK(name, level, delay, x)
X_DEFINE_ENUM_WITH_STRING_CONVERSIONS_TOSTRING_CASE(r, data, elem)
DEFINE_ENUM_WITH_STRING_CONVERSIONS(name, enumerators)
```

Defines

```
PERFORMANCE_COUNTER_ENABLED
namespace hpx

namespace parcelset

template<typename T>
struct performance_counter<T, false>
```

Public Functions

```
constexpr performance_counter()
constexpr performance_counter(const T&)
constexpr operator T() const
constexpr T operator=(const T&)
constexpr T operator++()
constexpr T operator++(int)
constexpr T operator+=(const T&)
constexpr T operator--()
constexpr T operator--(int)
constexpr T operator==(const T&)
```

Friends

```
std::ostream &operator<<(std::ostream &os, const performance_counter<T, false>&)

template<typename T>
struct performance_counter<T, true>
```

Public Functions

```
performance_counter()
performance_counter(const T &init)
operator T() const
T operator=(const T &x)
T operator++()
T operator++(int x)
```

```
T operator+=(const T &rhs)
T operator--()
T operator--(int x)
T operator==(const T &rhs)
```

Public Members

```
std::atomic<T> value_
```

Friends

```
std::ostream &operator<<(std::ostream &os, const performance_counter<T, true> &x)
namespace hpx
```

```
namespace parcelset
```

```
namespace policies
```

```
namespace libfabric
```

```
template<typename T, int Offset, typename Region, typename Allocator>
class pinned_memory_vector
```

Public Types

```
typedef T value_type
typedef value_type &reference
typedef T *iterator
typedef T const *const_iterator
typedef std::vector<T>::difference_type difference_type
typedef std::vector<T>::size_type size_type
typedef Allocator allocator_type
typedef Region region_type
typedef pinned_memory_vector<T, Offset, region_type, allocator_type> vector_type
typedef util::function_nonser<void ()> deleter_callback
```

Public Functions

```

pinned_memory_vector (allocator_type *alloc)
pinned_memory_vector (T *p, std::size_t s, deleter_callback cb, allocator_type *al-
loc, region_type *r)
pinned_memory_vector (vector_type &&other)
~pinned_memory_vector ()

vector_type &operator= (vector_type &&other)

size_type size () const
size_type max_size () const
bool empty () const

T *data ()

iterator begin ()
iterator end ()
const_iterator begin () const
const_iterator end () const
reference operator[] (std::size_t index)
const_reference operator[] (std::size_t index) const
void push_back (const T &_Val)
std::size_t capacity ()
void resize (std::size_t s)
void reserve (std::size_t s)

```

Public Members

```

const typedef value_type& hpx::parcelset::policies::libfabric::pinned_memor
T *m_array
std::size_t m_size
deleter_callback m_cb
allocator_type *m_alloc
region_type *m_region

```

Private Functions

```
pinned_memory_vector (vector_type const &other)

namespace hpx

namespace parcelset

namespace policies

namespace libfabric
```

TypeDefs

```
template<typename Mutex>
using scoped_lock = std::lock_guard<Mutex>

template<typename Mutex>
using unique_lock = std::unique_lock<Mutex>
```

Defines

```
RWL_DEBUG_MSG (x)
atomic_xadd (P, V)
cmpxchg (P, O, N)
atomic_inc (P)
rwl_barrier_()
namespace hpx

namespace lcos
```

```
namespace local

class readers_writer_mutex
```

Public Functions

```
readers_writer_mutex()

void lock()
void unlock()
bool try_lock()
void lock_shared()
```

```
void unlock_shared()  
bool try_lock_shared()  
bool owns_lock()
```

Private Members

```
readwrite_ticket ticket  
bool readlock_  
union readwrite_ticket
```

Public Members

```
uint64_t u  
uint32_t wr  
struct hpx::lcos::local::readers_writer_mutex::readwrite_ticket::[anonymous] i  
uint16_t writers  
uint16_t readers  
uint16_t next  
uint16_t notused  
struct hpx::lcos::local::readers_writer_mutex::readwrite_ticket::[anonymous] s
```

```
namespace hpx
```

```
namespace parcelset
```

```
namespace policies
```

```
namespace libfabric
```

```
struct receiver
```

Public Types

```
typedef libfabric_region_provider region_provider  
typedef rma_memory_region<region_provider> region_type  
typedef hpx::detail::small_vector<region_type*, 8> zero_copy_vector
```

Public Functions

```
receiver (parcelport *pp, fid_ep *endpoint, rma_memory_pool<region_provider> &memory_pool)  
receiver (receiver &&other)  
receiver &operator= (receiver &&other)  
~receiver ()  
void handle_recv (fi_addr_t const &src_addr, std::uint64_t len)  
void pre_post_receive ()  
void cleanup ()
```

Private Types

```
typedef hpx::lcos::local::spinlock mutex_type
```

Private Members

```
parcelport *pp_  
fid_ep *endpoint_  
region_type *header_region_  
rma_memory_pool<region_provider> *memory_pool_  
performance_counter<unsigned int> messages_handled_  
performance_counter<unsigned int> acks_received_  
performance_counter<unsigned int> msg_plain_  
performance_counter<unsigned int> msg_rma_  
performance_counter<unsigned int> sent_ack_  
performance_counter<unsigned int> rma_reads_  
performance_counter<unsigned int> recv_deletes_  
boost::lockfree::stack<rma_receiver*, boost::lockfree::capacity<HPX_PARCELPORT_LIBFABRIC_MAX_PR>  
mutex_type active_receivers_mtx_  
hpx::lcos::local::detail::condition_variable active_receivers_cv_  
hpx::util::atomic_count active_receivers_
```

Friends

```

friend hpx::parcelset::policies::libfabric::libfabric_controller
namespace hpx

namespace parcelset

namespace policies

namespace libfabric

struct rma_base
    Subclassed by hpx::parcelset::policies::libfabric::rma_receiver,
hpx::parcelset::policies::libfabric::sender

```

Public Functions

```

rma_base()
virtual ~rma_base()
virtual void handle_error(struct fi_cq_err_entry err) = 0

```

Defines

```

RDMA_POOL_1K_CHUNK_SIZE
RDMA_POOL_SMALL_CHUNK_SIZE
RDMA_POOL_MEDIUM_CHUNK_SIZE
RDMA_POOL_LARGE_CHUNK_SIZE
RDMA_POOL_MAX_1K_CHUNKS
RDMA_POOL_MAX_SMALL_CHUNKS
RDMA_POOL_MAX_MEDIUM_CHUNKS
RDMA_POOL_MAX_LARGE_CHUNKS
namespace hpx

namespace parcelset

template<typename RegionProvider>
struct memory_region_allocator

```

Public Types

```
typedef RegionProvider::provider_domain domain_type  
typedef rma_memory_region<RegionProvider> region_type  
typedef std::shared_ptr<region_type> region_ptr
```

Public Functions

```
memory_region_allocator()
```

Public Static Functions

```
static region_ptr malloc(domain_type *pd, const std::size_t bytes)  
static void free(region_ptr region)  
  
template<typename RegionProvider, typename Allocator, typename PoolType, std::size_t ChunkSize, std::size_t  
struct pool_container
```

Public Types

```
typedef RegionProvider::provider_domain domain_type  
typedef rma_memory_region<RegionProvider> region_type  
typedef std::shared_ptr<region_type> region_ptr
```

Public Functions

```
pool_container(domain_type *pd)  
bool allocate_pool()  
void DeallocatePool()  
void push(region_type *region)  
region_type *pop()  
void decrement_used_count(uint32_t N)  
std::string status()  
constexpr std::size_t chunk_size() const
```

Public Members

```
performance_counter<unsigned int> accesses_
performance_counter<unsigned int> in_use_
domain_type *pd_
std::unordered_map<const char*, region_ptr> block_list_
std::array<region_type, MaxChunks> region_list_
bl::stack<region_type*>, bl::capacity<MaxChunks>> free_list_

struct pool_large
```

Public Static Functions

```
static const char *desc()
struct pool_medium
```

Public Static Functions

```
static const char *desc()
struct pool_small
```

Public Static Functions

```
static const char *desc()
struct pool_tiny
```

Public Static Functions

```
static const char *desc()
template<typename RegionProvider>
struct rma_memory_pool
```

Public Types

```
typedef RegionProvider::provider_domain domain_type
typedef rma_memory_region<RegionProvider> region_type
typedef memory_region_allocator<RegionProvider> allocator_type
typedef std::shared_ptr<region_type> region_ptr
```

Public Functions

```
HPX_NON_COPYABLE (rma_memory_pool)

rma_memory_pool (domain_type *pd)
~rma_memory_pool ()
void deallocate_pools ()
bool can_allocate_unsafe (size_t length) const
region_type *allocate_region (size_t length)
void deallocate (region_type *region)
region_type *allocate_temporary_region (std::size_t length)
```

Public Members

```
domain_type *protection_domain_
pool_container<RegionProvider, allocator_type, pool_tiny, RDMA_POOL_1K_CHUNK_SIZE, RDMA_POOL_MAX>
pool_container<RegionProvider, allocator_type, pool_small, RDMA_POOL_SMALL_CHUNK_SIZE, RDMA_POOL_MAX>
pool_container<RegionProvider, allocator_type, pool_medium, RDMA_POOL_MEDIUM_CHUNK_SIZE, RDMA_POOL_MAX>
pool_container<RegionProvider, allocator_type, pool_large, RDMA_POOL_LARGE_CHUNK_SIZE, RDMA_POOL_MAX>
hpx::util::atomic_count temp_regions
hpx::util::atomic_count user_regions

namespace hpx
```



```
namespace parcelset
```

```
struct rma_memory_base
```

Public Functions

```
rma_memory_base ()

template<typename RegionProvider>
struct rma_memory_region
```

Public Types

```
enum [anonymous]
  Values:
    BLOCK_USER = 1
    BLOCK_TEMP = 2
    BLOCK_PARTIAL = 4
typedef RegionProvider::provider_domain provider_domain
typedef RegionProvider::provider_region provider_region
```

Public Functions

```
rma_memory_region()
rma_memory_region(provider_region *region, char *address, char *base_address, uint64_t
                     size, uint32_t flags)
rma_memory_region(provider_domain *pd, const void *buffer, const uint64_t length)
int allocate(provider_domain *pd, uint64_t length)
~rma_memory_region()
int release(void)
char *get_address(void) const
char *get_base_address(void) const
uint64_t get_size(void) const
void *get_desc(void) const
uint64_t get_remote_key(void) const
void set_message_length(uint32_t length)
uint32_t get_message_length(void) const
struct fid_mr *get_region()
void set_user_region()
bool get_user_region() const
void set_temp_region()
bool get_temp_region() const
void set_partial_region()
bool get_partial_region() const
```

Private Members

```
provider_region *region_
char *address_
char *base_addr_
uint64_t size_
uint64_t used_space_
uint32_t flags_
```

Friends

```
std::ostream &operator<< (std::ostream &os, rma_memory_region const &region)
namespace hpx
```

```
namespace traits
```

```
template<typename RegionProvider>
struct rma_memory_region_traits
```

Public Types

```
typedef RegionProvider::provider_domain provider_domain
typedef RegionProvider::provider_region provider_region
```

Public Static Functions

```
static int register_memory(provider_domain *pd, const void *buf, size_t len, uint64_t
                           access, uint64_t offset, uint64_t requested_key, uint64_t
                           flags, provider_region **mr, void *context)
static int unregister_memory(provider_region *mr)
static int flags()

namespace hpx

namespace parcelset

namespace policies

namespace libfabric

struct rma_receiver : public hpx::parcelset::policies::libfabric::rma_base
```

Public Types

```
typedef libfabric_region_provider region_provider
typedef rma_memory_region<region_provider> region_type
typedef rma_memory_pool<region_provider> memory_pool_type
typedef hpx::detail::small_vector<region_type*, 8> zero_copy_vector
typedef header<HPX_PARCELPORT_LIBFABRIC_MESSAGE_HEADER_SIZE> header_type
typedef serialization::serialization_chunk chunk_struct
typedef hpx::util::function_nonser<void (rma_receiver*)> completion_handler
```

Public Functions

```
rma_receiver (parcelport *pp, fid_ep *endpoint, memory_pool_type *memory_pool,
completion_handler &&handler)

~rma_receiver()

void read_message (region_type *region, fi_addr_t const &src_addr)

void handle_message_no_rma ()

void handle_message_with_zerocopy_rma ()

void handle_message_no_chunk_data ()

int handle_chunks_read_message ()

void handle_rma_read_completion ()

void send_rdma_complete_ack ()

void cleanup_receive ()

void handle_error (struct fi_cq_err_entry err)

void read_chunk_list ()

void read_one_chunk (fi_addr_t src_addr, region_type *get_region, const void *re-
moteAddr, uint64_t rkey)
```

Public Static Attributes

```
constexpr unsigned int header_size = header_type::header_block_size
```

Private Members

```
parcelport *pp_
fid_ep *endpoint_
region_type *header_region_
region_type *chunk_region_
region_type *message_region_
header_type *header_
std::vector<chunk_struct> chunks_
zero_copy_vector rma_regions_
rma_memory_pool<region_provider> *memory_pool_
fi_addr_t src_addr_
completion_handler handler_
hpx::util::atomic_count rma_count_
bool chunk_fetch_
double start_time_
performance_counter<unsigned int> msg_plain_
performance_counter<unsigned int> msg_rma_
performance_counter<unsigned int> sent_ack_
performance_counter<unsigned int> rma_reads_
performance_counter<unsigned int> recv_deletes_
```

Friends

```
friend hpx::parcelset::policies::libfabric::receiver
namespace hpx

namespace parcelset

namespace policies

namespace libfabric

struct sender : public hpx::parcelset::policies::libfabric::rma_base
```

Public Types

```
typedef libfabric_region_provider region_provider
typedef rma_memory_region<region_provider> region_type
typedef rma_memory_pool<region_provider> memory_pool_type
typedef header<HPX_PARCELPORT_LIBFABRIC_MESSAGE_HEADER_SIZE> header_type
typedef pinned_memory_vector<char, header_size, region_type, memory_pool_type> snd_data_type
typedef parcel_buffer<snd_data_type, serialization::serialization_chunk> snd_buffer_type
typedef hpx::detail::small_vector<region_type*, 8> zero_copy_vector
```

Public Functions

```
sender (parcelport *pp, fid_ep *endpoint, fid_domain *domain, memory_pool_type  
*memory_pool)
~sender ()

snd_buffer_type get_new_buffer ()

template<typename Handler, typename ParcelPostprocess>
void async_write (Handler&&, ParcelPostprocess&&)

void verify_ (parcelset::locality const&) const
void async_write_impl ()
void handle_send_completion ()
void handle_message_completion_ack ()
void cleanup ()
void handle_error (struct fi_cq_err_entry err)
```

Public Members

```
parcelport *parcelport_
fid_ep *endpoint_
fid_domain *domain_
memory_pool_type *memory_pool_
fi_addr_t dst_addr_
snd_buffer_type buffer_
region_type *header_region_
region_type *chunk_region_
region_type *message_region_
header_type *header_
zero_copy_vector rma_regions_
```

```
    hpx::util::atomic_count completion_count_
    performance_counter<unsigned int> sends_posted_
    performance_counter<unsigned int> sends_deleted_
    performance_counter<unsigned int> acks_received_
    util::unique_function_nonser<void (error_code const&) > handler_
    util::function_nonser<void (sender*)> postprocess_handler_
struct iovec region_list_[2]
    void *desc_[2]
```

Public Static Attributes

```
constexpr unsigned int header_size = header_type::header_block_size
```

```
namespace hpx
```

```
namespace concurrent
```

```
template<class Key, class Value, class Hash = std::hash<Key>, class KeyEqual = std::equal_to<Key>, class Allocator = std::allocator<Value>
class unordered_map
```

Public Types

```
typedef hpx::lcos::local::readers_writer_mutex rw_mutex_type
typedef std::unique_lock<rw_mutex_type> write_lock
typedef std::shared_lock<rw_mutex_type> read_lock
typedef std::defer_lock_t defer_lock
typedef base_map::key_type key_type
typedef base_map::mapped_type mapped_type
typedef base_map::value_type value_type
typedef base_map::size_type size_type
typedef base_map::difference_type difference_type
typedef base_map::hasher hasher
typedef base_map::key_equal key_equal
typedef base_map::allocator_type allocator_type
typedef base_map::reference reference
typedef base_map::const_reference const_reference
typedef base_map::pointer pointer
typedef base_map::const_pointer const_pointer
typedef base_map::iterator iterator
typedef base_map::const_iterator const_iterator
```

```

typedef base_map::local_iterator local_iterator
typedef base_map::const_local_iterator const_local_iterator
typedef read_lock map_read_lock_type
typedef write_lock map_write_lock_type

```

Public Functions

```

unordered_map (size_type n = 64, const hasher &hf = hasher(), const key_equal &eq = key_equal(), const allocator_type &a = allocator_type())
template<typename InputIterator>
unordered_map (InputIterator first, InputIterator last, size_type n = 64, const hasher &hf = hasher(), const key_equal &eq = key_equal(), const allocator_type &a = allocator_type())
unordered_map (const unordered_map &other)
unordered_map (const allocator_type &a)
unordered_map (const unordered_map &other, const allocator_type &a)
unordered_map (unordered_map &&other)
unordered_map (unordered_map &&other, const allocator_type &a)
unordered_map (std::initializer_list<value_type> il, size_type n = 64, const hasher &hf = hasher(), const key_equal &eq = key_equal(), const allocator_type &a = allocator_type())
~unordered_map ()

unordered_map &operator= (const unordered_map &other)
unordered_map &operator= (unordered_map &&other)
unordered_map &operator= (std::initializer_list<value_type> il)
void swap (unordered_map &other)
std::pair<iterator, bool> insert (const value_type &x)
iterator insert (const_iterator hint, const value_type &x)

template<class InputIterator>
void insert (InputIterator first, InputIterator last)
std::pair<iterator, bool> insert (value_type &&x)
iterator insert (const_iterator hint, value_type &&x)
void insert (std::initializer_list<value_type> il)

template<typename ...Args>
std::pair<iterator, bool> emplace (Args&&... args)

template<typename ...Args>
iterator emplace_hint (const_iterator hint, Args&&... args)

```

```
iterator erase (const_iterator position)
size_type erase (const key_type &k)
iterator erase (const_iterator first, const_iterator last)
void clear ()
bool empty () const
size_type size () const
size_type max_size () const
iterator begin ()
const_iterator begin () const
iterator end ()
const_iterator end () const
const_iterator cbegin () const
const_iterator cend () const
rw_mutex_type &read_write_mutex ()
hasher hash_function () const
key_equal key_eq () const
iterator find (const key_type &k)
const_iterator find (const key_type &k) const
std::pair<const_iterator, bool> is_in_map (const key_type &k) const
size_type count (const key_type &k) const
std::pair<iterator, iterator> equal_range (const key_type &k)
std::pair<const_iterator, const_iterator> equal_range (const key_type &k) const
const mapped_type &operator[] (const key_type &k) const
mapped_type &at (const key_type &k)
const mapped_type &at (const key_type &k) const
```

Private Types

```
typedef std::unordered_map<Key, Value, Hash, KeyEqual, Allocator> base_map
```

Private Members

```
base_map map_
rw_mutex_type mutex_
read_lock iterator_lock_
```

parcelport_mpi

See *Public API* for a list of names and headers that are part of the public HPX API.

parcelport_tcp

See *Public API* for a list of names and headers that are part of the public HPX API.

parcelset

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_ACTIONUSESMESSAGECOALESCINGDECLARATION(...)
HPX_ACTIONUSESMESSAGECOALESCINGDEFINITION(...)
HPX_ACTIONUSESMESSAGECOALESCING(...)
HPX_ACTIONUSESMESSAGECOALESCINGNOTHROWDECLARATION(...)
HPX_ACTIONUSESMESSAGECOALESCINGNOTHROWDEFINITION(...)
HPX_ACTIONUSESMESSAGECOALESCINGNOTHROW(...)
```

parcelset_base

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace parcelset
```

TypeDefs

```
using endpoints_type = std::map<std::string, locality>
```

Functions

```
std::ostream &operator<< (std::ostream &os, endpoints_type const &endpoints)  
class locality
```

Public Functions

```
locality()  
  
template<typename Impl, typename Enable1 = std::enable_if_t<!std::is_same_v<locality, std::decay_t<Impl>>>, ty  
locality(Impl &&i)  
  
locality(locality const &other)  
  
locality(locality &&other)  
  
locality &operator=(locality const &other)  
  
locality &operator=(locality &&other)  
  
operator bool() const  
  
char const *type() const  
  
template<typename Impl>  
Impl &get()  
  
template<typename Impl>  
Impl const &get() const
```

Private Functions

```
void save(serialization::output_archive &ar, const unsigned int version) const  
void load(serialization::input_archive &ar, const unsigned int version)  
Hpx_serialization_split_member()
```

Private Members

```
std::unique_ptr<impl_base> impl_
```

Friends

```
friend hpx::parcelset::hpx::serialization::access  
bool operator==(locality const &lhs, locality const &rhs)  
bool operator!=(locality const &lhs, locality const &rhs)  
bool operator<(locality const &lhs, locality const &rhs)  
bool operator>(locality const &lhs, locality const &rhs)
```

```
std::ostream &operator<< (std::ostream &os, locality const &l)
template<typename Impl>
class impl
```

Public Functions

```
impl (Impl &&i)
impl (Impl const &i)
bool equal (impl_base const &rhs) const
bool less_than (impl_base const &rhs) const
bool valid () const
const char *type () const
std::ostream &print (std::ostream &os) const
void save (serialization::output_archive &ar) const
void load (serialization::input_archive &ar)
impl_base *clone () const
impl_base *move ()
```

Public Members

```
Impl impl_
class impl_base
```

Public Functions

```
virtual ~impl_base ()
virtual bool equal (impl_base const &rhs) const = 0
virtual bool less_than (impl_base const &rhs) const = 0
virtual bool valid () const = 0
virtual const char *type () const = 0
virtual std::ostream &print (std::ostream &os) const = 0
virtual void save (serialization::output_archive &ar) const = 0
virtual void load (serialization::input_archive &ar) = 0
virtual impl_base *clone () const = 0
virtual impl_base *move () = 0
```

```
template<typename Impl>
Impl &get()

template<typename Impl>
Impl const &get() const

namespace hpx
```

```
namespace parcelset
```

Typedefs

```
using parcel_write_handler_type = util::function_nonser<void (std::error_code
                                                       const&,
                                                       parcelset::parcel
                                                       const&) >
```

The type of a function that can be registered as a parcel write handler using the function `hpx::set_parcel_write_handler`.

Note A parcel write handler is a function which is called by the parcel layer whenever a parcel has been sent by the underlying networking library and if no explicit parcel handler function was specified for the parcel.

Enums

```
enum parcelport_background_mode
```

Type of background work to perform.

Values:

```
parcelport_background_mode_flush_buffers = 0x01
    perform buffer flush operations
```

```
parcelport_background_mode_send = 0x03
    perform send operations (includes buffer flush)
```

```
parcelport_background_mode_receive = 0x04
    perform receive operations
```

```
parcelport_background_mode_all = 0x07
    perform all operations
```

performance_counters

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace performance_counters
```

Functions

```
bool action_invocation_counter_discoverer(hpx::actions::detail::invocation_count_registry
                                         const &registry, counter_info const
                                         &info, counter_path_elements &p,
                                         discover_counter_func const &f, discover_counters_mode mode, error_code
                                         &ec)
```

namespace hpx

namespace performance_counters

Functions

```
void register_agas_counter_types(agas::addressing_service &client)
```

Install performance counter types exposing properties from the local cache.

namespace hpx

namespace agas

Enums

```
enum namespace_action_code
```

Values:

```
invalid_request = 0
locality_ns_service = 0b1100000
locality_ns_bulk_service = 0b1100001
locality_ns_allocate = 0b1100010
locality_ns_free = 0b1100011
locality_ns_localities = 0b1100100
locality_ns_num_localities = 0b1100101
locality_ns_num_threads = 0b1100110
locality_ns_statistics_counter = 0b1100111
locality_ns_resolve_locality = 0b1101000
primary_ns_service = 0b1000000
primary_ns_bulk_service = 0b1000001
primary_ns_route = 0b1000010
primary_ns_bind_gid = 0b1000011
primary_ns_resolve_gid = 0b1000100
primary_ns_unbind_gid = 0b1000101
primary_ns_increment_credit = 0b1000110
```

```
primary_ns_decrement_credit = 0b1000111
primary_ns_allocate = 0b1001000
primary_ns_begin_migration = 0b1001001
primary_ns_end_migration = 0b1001010
primary_ns_statistics_counter = 0b1001011
component_ns_service = 0b0100000
component_ns_bulk_service = 0b0100001
component_ns_bind_prefix = 0b0100010
component_ns_bind_name = 0b0100011
component_ns_resolve_id = 0b0100100
component_ns_unbind_name = 0b0100101
component_ns_iterate_types = 0b0100110
component_ns_get_component_type_name = 0b0100111
component_ns_num_localities = 0b0101000
component_ns_statistics_counter = 0b0101001
symbol_ns_service = 0b0010000
symbol_ns_bulk_service = 0b0010001
symbol_ns_bind = 0b0010010
symbol_ns_resolve = 0b0010011
symbol_ns_unbind = 0b0010100
symbol_ns_iterate_names = 0b0010101
symbol_ns_on_event = 0b0010110
symbol_ns_statistics_counter = 0b0010111
```

Variables

```
constexpr char const *const performance_counter_basename = "/agas/"

namespace hpx

namespace performance_counters

template<typename Derived>
class base_performance_counter
```

Public Types

```
typedef Derived type_holder  
typedef hpx::performance_counters::server::base_performance_counter base_type_holder
```

Public Functions

```
base_performance_counter()  
base_performance_counter(hpx::performance_counters::counter_info const &info)  
void finalize()  
hpx::naming::address get_current_address() const
```

Private Types

```
    typedef hpx::components::component_base<Derived> base_type  
namespace hpx  
  
namespace aqas
```

Functions

```
void component_namespace_register_counter_types(error_code &ec = throws)
```

namespace performance_counters

Functions

```
bool default_counter_discoverer(counter_info const&, discover_counter_func const&, discover_counters_mode, error_code&)
```

Default discovery function for performance counters; to be registered with the counter types. It will pass the `counter_info` and the `error_code` to the supplied function.

```
bool locality_counter_discoverer(counter_info const&, discover_counter_func const&, discover_counters_mode, error_code&)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/total)/<instancename>

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/pool#<pool_name>/total)/<instancename>

```
bool locality0_counter_discoverer(counter_info const&, discover_counter_func const&, discover_counters_mode, error_code&)
```

Default discoverer function for AGAS performance counters; to be registered with the counter types.
It is suitable to be used for all counters following the naming scheme:

```
/<objectname>{locality#0/total}<instancename>
```

```
bool locality_thread_counter_discoverer(counter_info const&, discover_counter_func const&, discover_counters_mode, error_code&)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

```
/<objectname>(locality#<locality_id>/worker-thread#<threadnum>)<instancename>
```

```
bool locality_pool_thread_counter_discoverer(counter_info const &info, discover_counter_func const &f, discover_counters_mode mode, error_code &ec)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

```
/<objectname>{locality#<locality_id>/pool#<poolname>/thread#<threadnum>}<instancename>
```

```
bool locality_pool_thread_no_total_counter_discoverer(counter_info const &info, discover_counter_func const &f, discover_counters_mode mode, error_code &ec)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

```
/<objectname>{locality#<locality_id>/pool#<poolname>/thread#<threadnum>}<instancename>
```

This is essentially the same as above just that locality#*/total is not supported.

```
bool locality numa_counter_discoverer(counter_info const&, discover_counter_func const&, discover_counters_mode, error_code&)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

```
/<objectname>(locality#<locality_id>/numa-node#<threadnum>)<instancename>
```

```
naming::gid_type locality_raw_counter_creator(counter_info const&, hpx::util::function_nonser<std::int64_t> bool > const&, error_code& Creation function for raw counters. The passed function is encapsulating the actual value to monitor. This function checks the validity of the supplied counter name, it has to follow the scheme:
```

```
/<objectname>(locality#<locality_id>/total)<instancename>
```

```
naming::gid_type locality_raw_values_counter_creator(counter_info const&, hpx::util::function_nonser<std::vector<std::int64_t>> const&, error_code&
```

```
naming::gid_type agas_raw_counter_creator(counter_info const&, error_code&, char const*const)
```

Creation function for raw counters. The passed function is encapsulating the actual value to monitor. This function checks the validity of the supplied counter name, it has to follow the scheme:

```

/agas(<objectinstance>/total)/<instancename>

bool agas_counter_discoverer(counter_info const&, discover_counter_func const&,
                               discover_counters_mode, error_code&)
    Default discoverer function for performance counters; to be registered with the counter types. It is
    suitable to be used for all counters following the naming scheme:
/agas(<objectinstance>/total)/<instancename>

naming::gid_type local_action_invocation_counter_creator(counter_info
                                                               const&,           er-
                                                               ror_code&)

bool local_action_invocation_counter_discoverer(counter_info const&, dis-
                                                       cover_counter_func const&,
                                                       discover_counters_mode,
                                                       error_code&)

namespace hpx

namespace performance_counters

Functions

hpx::future<id_type> create_performance_counter_async(id_type           target_id,
                                                               counter_info const &info)

id_type create_performance_counter(id_type target_id, counter_info const &info, er-
                                         or_code &ec = throws)

namespace hpx

```

```
namespace performance_counters
```

Functions

```

bool parse_counter_name(std::string const &name, path_elements &elements)

struct instance_elements

```

Public Members

```

instance_name parent_
instance_name child_
instance_name subchild_

struct instance_name

```

Public Members

```
    std::string name_
    std::string index_
    bool basename_ = false

struct path_elements
```

Public Members

```
    std::string object_
    instance_elements instance_
    std::string counter_
    std::string parameters_
```

```
namespace hpx
```

```
namespace performance_counters
```

TypeDefs

```
typedef hpx::util::function_nonser<naming::gid_type (counter_info const&, error_code&) >
    create_counter_func
```

This declares the type of a function, which will be called by HPX whenever a new performance counter instance of a particular type needs to be created.

```
typedef hpx::util::function_nonser<bool (counter_info const&, error_code&) >
    discover_counter_func
```

This declares a type of a function, which will be passed to a *discover_counters_func* in order to be called for each discovered performance counter instance.

```
typedef hpx::util::function_nonser<bool (counter_info const&, discover_counter_func
    const&, discover_counters_mode, error_code&) >
    discover_counters_func
```

This declares the type of a function, which will be called by HPX whenever it needs to discover all performance counter instances of a particular type.

Enums

```
enum counter_type
```

Values:

```
counter_text
```

counter_text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

```
counter_raw
```

counter_raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

```
counter_monotonically_increasing
```

counter_average_base

counter_average_base is used as the base data (denominator) in the computation of time or count averages for the *counter_average_count* and *counter_average_timer* counter types. This counter type collects the last observed value only.

Formula: None. This counter uses raw data in factorial calculations without delivering an output.
Average: $\text{SUM} (N) / x$ Type: Instantaneous

counter_average_count

counter_average_count shows how many items are processed, on average, during an operation. Counters of this type display a ratio of the items processed (such as bytes sent) to the number of operations completed. The ratio is calculated by comparing the number of items processed during the last interval to the number of operations completed during the last interval.

Formula: $(N_1 - N_0) / (D_1 - D_0)$, where the numerator (N) represents the number of items processed during the last sample interval, and the denominator (D) represents the number of operations completed during the last two sample intervals. Average: $(N_x - N_0) / (D_x - D_0)$ Type: Average

counter_aggregating

counter_aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: $F(N_x)$

counter_average_timer

counter_average_timer measures the average time it takes to complete a process or operation. Counters of this type display a ratio of the total elapsed time of the sample interval to the number of processes or operations completed during that time. This counter type measures time in ticks of the system clock. The variable F represents the number of ticks per second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $((N_1 - N_0) / F) / (D_1 - D_0)$, where the numerator (N) represents the number of ticks counted during the last sample interval, the variable F represents the frequency of the ticks, and the denominator (D) represents the number of operations completed during the last sample interval. Average: $((N_x - N_0) / F) / (D_x - D_0)$ Type: Average

counter_elapsed_time

counter_elapsed_time shows the total time between when the component or process started and the time when this value is calculated. The variable F represents the number of time units that elapse in one second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $(D_0 - N_0) / F$, where the nominator (D) represents the current time, the numerator (N) represents the time the object was started, and the variable F represents the number of time units that elapse in one second. Average: $(D_x - N_0) / F$ Type: Difference

counter_histogram

counter_histogram exposes a histogram of the measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a *counter_value*. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

The first three values in the returned array represent the lower and upper boundaries, and the size of the histogram buckets. All remaining values in the returned array represent the number of measurements for each of the buckets in the histogram.

counter_raw_values

counter_raw_values exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a

`counter_value`. Those will also not implement the `get_counter_value()` functionality. The results are exposed through a separate `get_counter_values_array()` function.

counter_text

`counter_text` shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

counter_raw

`counter_raw` shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

counter_monotonically_increasing**counter_average_base**

`counter_average_base` is used as the base data (denominator) in the computation of time or count averages for the `counter_average_count` and `counter_average_timer` counter types. This counter type collects the last observed value only.

Formula: None. This counter uses raw data in factional calculations without delivering an output.
Average: SUM (N) / x Type: Instantaneous

counter_average_count

`counter_average_count` shows how many items are processed, on average, during an operation. Counters of this type display a ratio of the items processed (such as bytes sent) to the number of operations completed. The ratio is calculated by comparing the number of items processed during the last interval to the number of operations completed during the last interval.

Formula: $(N_1 - N_0) / (D_1 - D_0)$, where the numerator (N) represents the number of items processed during the last sample interval, and the denominator (D) represents the number of operations completed during the last two sample intervals. Average: $(N_x - N_0) / (D_x - D_0)$ Type: Average

counter_aggregating

`counter_aggregating` applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: $F(N_x)$

counter_average_timer

`counter_average_timer` measures the average time it takes to complete a process or operation. Counters of this type display a ratio of the total elapsed time of the sample interval to the number of processes or operations completed during that time. This counter type measures time in ticks of the system clock. The variable F represents the number of ticks per second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $((N_1 - N_0) / F) / (D_1 - D_0)$, where the numerator (N) represents the number of ticks counted during the last sample interval, the variable F represents the frequency of the ticks, and the denominator (D) represents the number of operations completed during the last sample interval.
Average: $((N_x - N_0) / F) / (D_x - D_0)$ Type: Average

counter_elapsed_time

`counter_elapsed_time` shows the total time between when the component or process started and the time when this value is calculated. The variable F represents the number of time units that elapse in one second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $(D_0 - N_0) / F$, where the nominator (D) represents the current time, the numerator (N) represents the time the object was started, and the variable F represents the number of time units that elapse in one second. Average: $(D_x - N_0) / F$ Type: Difference

counter_histogram

counter_histogram exposes a histogram of the measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a *counter_value*. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

The first three values in the returned array represent the lower and upper boundaries, and the size of the histogram buckets. All remaining values in the returned array represent the number of measurements for each of the buckets in the histogram.

counter_raw_values

counter_raw_values exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a *counter_value*. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

enum counter_status

Status and error codes used by the functions related to performance counters.

Values:

status_valid_data

No error occurred, data is valid.

status_new_data

Data is valid and different from last call.

status_invalid_data

Some error occurred, data is not value.

status_already_defined

The type or instance already has been defined.

status_counter_unknown

The counter instance is unknown.

status_counter_type_unknown

The counter type is unknown.

status_generic_error

A unknown error occurred.

status_valid_data

No error occurred, data is valid.

status_new_data

Data is valid and different from last call.

status_invalid_data

Some error occurred, data is not value.

status_already_defined

The type or instance already has been defined.

status_counter_unknown

The counter instance is unknown.

status_counter_type_unknown

The counter type is unknown.

status_generic_error

A unknown error occurred.

Functions

```
std::string &ensure_counter_prefix(std::string &name)
std::string ensure_counter_prefix(std::string const &counter)
std::string &remove_counter_prefix(std::string &name)
std::string remove_counter_prefix(std::string const &counter)
char const *get_counter_type_name(counter_type state)
    Return the readable name of a given counter type.

bool status_is_valid(counter_status s)

counter_status add_counter_type(counter_info const &info, error_code &ec)
naming::id_type get_counter(std::string const &name, error_code &ec)
naming::id_type get_counter(counter_info const &info, error_code &ec)
```

Variables

```
constexpr const char counter_prefix[] = "/counters"
constexpr std::size_t counter_prefix_len = (sizeof(counter_prefix) / sizeof(counter_prefix[0])) - 1

struct counter_info
```

Public Functions

```
counter_info(counter_type type = counter_raw)
counter_info(std::string const &name)
counter_info(counter_type type, std::string const &name, std::string const &help-
    text = "", std::uint32_t version = HPX_PERFORMANCE_COUNTER_V1,
    std::string const &uom = "")
```

Public Members

counter_type type_
The type of the described counter.

std::uint32_t version_
The version of the described counter using the 0xMMmmSSSS scheme

counter_status status_
The status of the counter object.

std::string fullname_
The full name of this counter.

std::string helptext_
The full descriptive text for this counter.

std::string unit_of_measure_
The unit of measure for this counter.

Private Functions

```
void serialize(serialization::output_archive &ar, const unsigned int)
void serialize(serialization::input_archive &ar, const unsigned int)
```

Friends

```
friend hpx::performance_counters::hpx::serialization::access

struct counter_path_elements : public hpx::performance_counters::counter_type_path_elements
#include <counters.hpp> A counter_path_elements holds the elements of a full name for a counter instance. Generally, a full name of a counter instance has the structure:  

/objectname{parentinstancename::parentindex/instancename#instanceindex } /counter-name#parameters  

i.e. /queue{localityprefix/thread#2}/length
```

Public Types

```
typedef counter_type_path_elements base_type
```

Public Functions

```
counter_path_elements()

counter_path_elements(std::string const &objectname, std::string const &countername, std::string const &parameters, std::string const &parentname, std::string const &instancename, std::int64_t parentindex = -1, std::int64_t instanceindex = -1, bool parentinstance_is_basename = false)

counter_path_elements(std::string const &objectname, std::string const &countername, std::string const &parameters, std::string const &parentname, std::string const &instancename, std::string const &subinstancename, std::int64_t parentindex = -1, std::int64_t instanceindex = -1, std::int64_t subinstanceindex = -1, bool parentinstance_is_basename = false)
```

Public Members

```
std::string parentinstancename_
the name of the parent instance

std::string instancename_
the name of the object instance

std::string subinstancename_
the name of the object sub-instance

std::int64_t parentinstanceindex_
the parent instance index
```

```
std::int64_t instanceindex_
    the instance index

std::int64_t subinstanceindex_
    the sub-instance index

bool parentinstance_is_basename_
    the parentinstancename_
```

Private Functions

```
void serialize(serialization::output_archive &ar, const unsigned int)
void serialize(serialization::input_archive &ar, const unsigned int)
```

Friends

```
friend hpx::performance_counters::hpx::serialization::access
    member holds a base counter name

struct counter_type_path_elements
    #include <counters.hpp> A counter_type_path_elements holds the elements of a full name for a
    counter type. Generally, a full name of a counter type has the structure:
    /objectname/countername
    i.e. /queue/length
    Subclassed by hpx::performance_counters::counter_path_elements
```

Public Functions

```
counter_type_path_elements()
counter_type_path_elements(std::string const &objectname, std::string const
    &countername, std::string const &parameters)
```

Public Members

```
std::string objectname_
    the name of the performance object

std::string countername_
    contains the counter name

std::string parameters_
    optional parameters for the counter instance
```

Protected Functions

```
void serialize(serialization::output_archive &ar, const unsigned int)
void serialize(serialization::input_archive &ar, const unsigned int)
```

Friends

```
friend hpx::performance_counters::hpx::serialization::access
```

Defines

```
HPX_PERFORMANCE_COUNTER_V1
namespace hpx
```

```
namespace performance_counters
```

Enums

```
enum counter_type
```

Values:

counter_text

counter_text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

counter_raw

counter_raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

counter_monotonically_increasing

counter_average_base

counter_average_base is used as the base data (denominator) in the computation of time or count averages for the *counter_average_count* and *counter_average_timer* counter types. This counter type collects the last observed value only.

Formula: None. This counter uses raw data in factional calculations without delivering an output.
Average: SUM (N) / x Type: Instantaneous

counter_average_count

counter_average_count shows how many items are processed, on average, during an operation. Counters of this type display a ratio of the items processed (such as bytes sent) to the number of operations completed. The ratio is calculated by comparing the number of items processed during the last interval to the number of operations completed during the last interval.

Formula: (N1 - N0) / (D1 - D0), where the numerator (N) represents the number of items processed during the last sample interval, and the denominator (D) represents the number of operations completed during the last two sample intervals. Average: (Nx - N0) / (Dx - D0) Type: Average

counter_aggregating

counter_aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: $F(N_x)$

counter_average_timer

counter_average_timer measures the average time it takes to complete a process or operation. Counters of this type display a ratio of the total elapsed time of the sample interval to the number of processes or operations completed during that time. This counter type measures time in ticks of the system clock. The variable F represents the number of ticks per second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $((N_1 - N_0) / F) / (D_1 - D_0)$, where the numerator (N) represents the number of ticks counted during the last sample interval, the variable F represents the frequency of the ticks, and the denominator (D) represents the number of operations completed during the last sample interval.
Average: $((N_x - N_0) / F) / (D_x - D_0)$ Type: Average

counter_elapsed_time

counter_elapsed_time shows the total time between when the component or process started and the time when this value is calculated. The variable F represents the number of time units that elapse in one second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $(D_0 - N_0) / F$, where the nominator (D) represents the current time, the numerator (N) represents the time the object was started, and the variable F represents the number of time units that elapse in one second. Average: $(D_x - N_0) / F$ Type: Difference

counter_histogram

counter_histogram exposes a histogram of the measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a *counter_value*. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

The first three values in the returned array represent the lower and upper boundaries, and the size of the histogram buckets. All remaining values in the returned array represent the number of measurements for each of the buckets in the histogram.

counter_raw_values

counter_raw_values exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a *counter_value*. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

counter_text

counter_text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

counter_raw

counter_raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

counter_monotonically_increasing**counter_average_base**

counter_average_base is used as the base data (denominator) in the computation of time or count averages for the *counter_average_count* and *counter_average_timer* counter types. This counter type collects the last observed value only.

Formula: None. This counter uses raw data in factional calculations without delivering an output.
 Average: $\text{SUM } (N) / x$ Type: Instantaneous

counter_average_count

counter_average_count shows how many items are processed, on average, during an operation. Counters of this type display a ratio of the items processed (such as bytes sent) to the number of operations completed. The ratio is calculated by comparing the number of items processed during the last interval to the number of operations completed during the last interval.

Formula: $(N1 - N0) / (D1 - D0)$, where the numerator (N) represents the number of items processed during the last sample interval, and the denominator (D) represents the number of operations completed during the last two sample intervals. Average: $(Nx - N0) / (Dx - D0)$ Type: Average

counter_aggregating

counter_aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: $F(Nx)$

counter_average_timer

counter_average_timer measures the average time it takes to complete a process or operation. Counters of this type display a ratio of the total elapsed time of the sample interval to the number of processes or operations completed during that time. This counter type measures time in ticks of the system clock. The variable F represents the number of ticks per second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $((N1 - N0) / F) / (D1 - D0)$, where the numerator (N) represents the number of ticks counted during the last sample interval, the variable F represents the frequency of the ticks, and the denominator (D) represents the number of operations completed during the last sample interval. Average: $((Nx - N0) / F) / (Dx - D0)$ Type: Average

counter_elapsed_time

counter_elapsed_time shows the total time between when the component or process started and the time when this value is calculated. The variable F represents the number of time units that elapse in one second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: $(D0 - N0) / F$, where the nominator (D) represents the current time, the numerator (N) represents the time the object was started, and the variable F represents the number of time units that elapse in one second. Average: $(Dx - N0) / F$ Type: Difference

counter_histogram

counter_histogram exposes a histogram of the measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a *counter_value*. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

The first three values in the returned array represent the lower and upper boundaries, and the size of the histogram buckets. All remaining values in the returned array represent the number of measurements for each of the buckets in the histogram.

counter_raw_values

counter_raw_values exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a *counter_value*. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

enum counter_status

Values:

```
status_valid_data
    No error occurred, data is valid.

status_new_data
    Data is valid and different from last call.

status_invalid_data
    Some error occurred, data is not value.

status_already_defined
    The type or instance already has been defined.

status_counter_unknown
    The counter instance is unknown.

status_counter_type_unknown
    The counter type is unknown.

status_generic_error
    A unknown error occurred.

status_valid_data
    No error occurred, data is valid.

status_new_data
    Data is valid and different from last call.

status_invalid_data
    Some error occurred, data is not value.

status_already_defined
    The type or instance already has been defined.

status_counter_unknown
    The counter instance is unknown.

status_counter_type_unknown
    The counter type is unknown.

status_generic_error
    A unknown error occurred.

enum discover_counters_mode
    Values:
        discover_counters_minimal
        discover_counters_full
```

Functions

```
counter_status get_counter_type_name(counter_type_path_elements const &path,
                                     std::string &result, error_code &ec = throws)
Create a full name of a counter type from the contents of the given counter_type_path_elements
instance.The generated counter type name will not contain any parameters.

counter_status get_full_counter_type_name(counter_type_path_elements const &path,
                                         std::string &result, error_code &ec =
                                         throws)
Create a full name of a counter type from the contents of the given counter_type_path_elements
instance. The generated counter type name will contain all parameters.
```

```
counter_status get_counter_name(counter_path_elements const &path, std::string &result,  
error_code &ec = throws)
```

Create a full name of a counter from the contents of the given counter_path_elements instance.

```
counter_status get_counter_instance_name(counter_path_elements const &path,  
std::string &result, error_code &ec = throws)
```

Create a name of a counter instance from the contents of the given counter_path_elements instance.

```
counter_status get_counter_type_path_elements(std::string const &name,  
counter_type_path_elements &path,  
error_code &ec = throws)
```

Fill the given counter_type_path_elements instance from the given full name of a counter type.

```
counter_status get_counter_path_elements(std::string const &name,  
counter_path_elements &path, error_code  
&ec = throws)
```

Fill the given counter_path_elements instance from the given full name of a counter.

```
counter_status get_counter_name(std::string const &name, std::string &countername, er-  
ror_code &ec = throws)
```

Return the canonical counter instance name from a given full instance name.

```
counter_status get_counter_type_name(std::string const &name, std::string &type_name,  
error_code &ec = throws)
```

Return the canonical counter type name from a given (full) instance name.

```
counter_status complement_counter_info(counter_info &info, counter_info const  
&type_info, error_code &ec = throws)
```

Complement the counter info if parent instance name is missing.

```
counter_status complement_counter_info(counter_info &info, error_code &ec = throws)
```

```
counter_status add_counter_type(counter_info const &info, create_counter_func const  
&create_counter, discover_counters_func const &dis-  
cover_counters, error_code &ec = throws)
```

```
counter_status discover_counter_types(discover_counter_func const &dis-  
cover_counter, discover_counters_mode mode  
= discover_counters_minimal, error_code &ec =  
throws)
```

Call the supplied function for each registered counter type.

```
counter_status discover_counter_types(std::vector<counter_info> &counters,  
discover_counters_mode mode = dis-  
cover_counters_minimal, error_code &ec =  
throws)
```

Return a list of all available counter descriptions.

```
counter_status discover_counter_type(std::string const &name, dis-  
cover_counter_func const &discover_counter,  
discover_counters_mode mode = dis-  
cover_counters_minimal, error_code &ec =  
throws)
```

Call the supplied function for the given registered counter type.

```
counter_status discover_counter_type(counter_info const &info, dis-  
cover_counter_func const &discover_counter,  
discover_counters_mode mode = dis-  
cover_counters_minimal, error_code &ec =  
throws)
```

```
counter_status discover_counter_type (std::string const &name,  
                                     std::vector<counter_info> const &counters,  
                                     discover_counters_mode mode = discover_counters_minimal,  
                                     error_code &ec = throws)
```

Return a list of matching counter descriptions for the given registered counter type.

```
counter_status discover_counter_type (counter_info const &info,  
                                     std::vector<counter_info> const &counters,  
                                     discover_counters_mode mode = discover_counters_minimal,  
                                     error_code &ec = throws)
```

```
bool expand_counter_info (counter_info const&, discover_counter_func const&, error_code&)
```

call the supplied function will all expanded versions of the supplied counter info.

This function expands all locality## and worker-thread## wild cards only.

```
counter_status remove_counter_type (counter_info const &info, error_code &ec = throws)
```

Remove an existing counter type from the (local) registry.

Note This doesn't remove existing counters of this type, it just inhibits defining new counters using this type.

```
counter_status get_counter_type (std::string const &name, counter_info &info, error_code  
&ec = throws)
```

Retrieve the counter type for the given counter name from the (local) registry.

```
hpx::future<naming::id_type> get_counter_async (std::string name, error_code &ec = throws)
```

Get the global id of an existing performance counter, if the counter does not exist yet, the function attempts to create the counter based on the given counter name.

```
hpx::future<naming::id_type> get_counter_async (counter_info const &info, error_code  
&ec = throws)
```

Get the global id of an existing performance counter, if the counter does not exist yet, the function attempts to create the counter based on the given counter info.

```
void get_counter_infos (counter_info const &info, counter_type &type, std::string &help-  
text, std::uint32_t &version, error_code &ec = throws)
```

Retrieve the meta data specific for the given counter instance.

```
void get_counter_infos (std::string name, counter_type &type, std::string &help-  
text, std::uint32_t &version, error_code &ec = throws)
```

Retrieve the meta data specific for the given counter instance.

```
struct counter_value
```

Public Functions

```
counter_value (std::int64_t value = 0, std::int64_t scaling = 1, bool scale_inverse = false)
template<typename T>
T get_value (error_code &ec = throws) const
    Retrieve the ‘real’ value of the counter_value, converted to the requested type T.
```

Public Members

std::uint64_t time_
 The local time when data was collected.

std::uint64_t count_
 The invocation counter for the data.

std::int64_t value_
 The current counter value.

std::int64_t scaling_
 The scaling of the current counter value.

counter_status status_
 The status of the counter value.

bool scale_inverse_
 If true, *value_* needs to be divided by *scaling_*, otherwise it has to be multiplied.

Private Functions

```
void serialize (serialization::output_archive &ar, const unsigned int)
```

```
void serialize (serialization::input_archive &ar, const unsigned int)
```

Friends

```
friend hpx::performance_counters::hpx::serialization::access
struct counter_values_array
```

Public Functions

```
counter_values_array (std::int64_t scaling = 1, bool scale_inverse = false)
counter_values_array (std::vector<std::int64_t> &&values, std::int64_t scaling = 1, bool
                            scale_inverse = false)
counter_values_array (std::vector<std::int64_t> const &values, std::int64_t scaling =
                            1, bool scale_inverse = false)

template<typename T>
T get_value (std::size_t index, error_code &ec = throws) const
    Retrieve the ‘real’ value of the counter_value, converted to the requested type T.
```

Public Members

`std::uint64_t time_`

The local time when data was collected.

`std::uint64_t count_`

The invocation counter for the data.

`std::vector<std::int64_t> values_`

The current counter values.

`std::int64_t scaling_`

The scaling of the current counter values.

`counter_status status_`

The status of the counter value.

`bool scale_inverse_`

If true, value_ needs to be divided by scaling_, otherwise it has to be multiplied.

Private Functions

`void serialize(serialization::output_archive &ar, const unsigned int)`

`void serialize(serialization::input_archive &ar, const unsigned int)`

Friends

```
friend hpx::performance_counters::hpx::serialization::access  
namespace hpx
```

`namespace agas`

Functions

```
void locality_namespace_register_counter_types(error_code &ec = throws)  
namespace hpx
```

`namespace performance_counters`

Functions

```
void install_counter(naming::id_type const &id, counter_info const &info, error_code  
&ec = throws)
```

Install a new performance counter in a way, which will uninstall it automatically during shutdown.

```
namespace hpx
```

`namespace performance_counters`

Functions

```
counter_status install_counter_type (std::string           const      &name,
                                     hpx::util::function_nonser<std::int64_t> bool
                                     > const &counter_value, std::string const &helptext = "", std::string const &uom = "",
                                     counter_type type = counter_raw, error_code &ec = throws)Install a new generic performance counter
type in a way, which will uninstall it automatically during shutdown.
```

The function *install_counter_type* will register a new generic counter type based on the provided function. The counter type will be automatically unregistered during system shutdown. Any consumer querying any instance of this counter type will cause the provided function to be called and the returned value to be exposed as the counter value.

The counter type is registered such that there can be one counter instance per locality. The expected naming scheme for the counter instances is: '/objectname{locality#<*>/total}/countername' where '<*>' is a zero based integer identifying the locality the counter is created on.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Return If successful, this function returns *status_valid_data*, otherwise it will either throw an exception or return an *error_code* from the enum *counter_status* (also, see note related to parameter *ec*).

Note The counter type registry is a locality based service. You will have to register each counter type on every locality where a corresponding performance counter will be created.

Parameters

- *name*: [in] The global virtual name of the counter type. This name is expected to have the format /objectname/countername.
- *counter_value*: [in] The function to call whenever the counter value is requested by a consumer.
- *helptext*: [in, optional] A longer descriptive text shown to the user to explain the nature of the counters created from this type.
- *uom*: [in] The unit of measure for the new performance counter type.
- *type*: [in] Type for the new performance counter type.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

```
counter_status install_counter_type (std::string           const      &name,
                                     hpx::util::function_nonser<std::vector<std::int64_t>> bool
                                     > const &counter_value, std::string const &helptext = "", std::string const &uom = "",
                                     error_code &ec = throws)Install a new generic performance counter type returning an array of values in
a way, that will uninstall it automatically during shutdown.
```

The function *install_counter_type* will register a new generic counter type that returns an array of values based on the provided function. The counter type will be automatically unregistered during system shutdown. Any consumer querying any instance of this 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, er-  
    ror_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 hpx
```

```
namespace performance_counters
```

Functions

```
std::vector<performance_counter> discover_counters (std::string const &name, error_code  
&ec = throws)  
struct performance_counter: public components::client_base<performance_counter, server::base_perfor
```

Public Types

```
using base_type = components::client_base<performance_counter, server::base_performance_counter>
```

Public Functions

```
performance_counter()  
performance_counter (std::string const &name)  
performance_counter (std::string const &name, hpx::id_type const &locality)  
performance_counter (id_type const &id)  
performance_counter (future<id_type> &&id)  
performance_counter (hpx::future<performance_counter> &&c)  
future<counter_info> get_info () const  
counter_info get_info (launch::sync_policy, error_code &ec = throws) const  
future<counter_value> get_counter_value (bool reset = false)  
counter_value get_counter_value (launch::sync_policy, bool reset = false, error_code  
&ec = throws)  
future<counter_value> get_counter_value () const  
counter_value get_counter_value (launch::sync_policy, error_code &ec = throws)  
future<counter_values_array> get_counter_values_array (bool reset = false)  
counter_values_array get_counter_values_array (launch::sync_policy, bool reset =  
false, error_code &ec = throws)  
future<counter_values_array> get_counter_values_array () const  
counter_values_array get_counter_values_array (launch::sync_policy, error_code,  
&ec = throws) const  
future<bool> start ()  
bool start (launch::sync_policy, error_code &ec = throws)  
future<bool> stop ()  
bool stop (launch::sync_policy, error_code &ec = throws)  
future<void> reset ()
```

```

void reset (launch::sync_policy, error_code &ec = throws)
future<void> reinit (bool reset = true)
void reinit (launch::sync_policy, bool reset = true, error_code &ec = throws)
future<std::string> get_name () const
std::string get_name (launch::sync_policy, error_code &ec = throws) const
template<typename T>
future<T> get_value (bool reset = false)
template<typename T>
T get_value (launch::sync_policy, bool reset = false, error_code &ec = throws)
template<typename T>
future<T> get_value () const
template<typename T>
T get_value (launch::sync_policy, error_code &ec = throws) const

```

Private Static Functions

```

template<typename T>
static T extract_value (future<counter_value> &&value)
namespace hpx

namespace performance_counters

struct performance_counter_base
Subclassed by hpx::performance_counters::server::base_performance_counter

```

Public Functions

```

virtual ~performance_counter_base()
Destructor, needs to be virtual to allow for clean destruction of derived objects

virtual counter_info get_counter_info () const = 0

virtual counter_value get_counter_value (bool reset = false) = 0

virtual counter_values_array get_counter_values_array (bool reset = false) = 0

virtual void reset_counter_value () = 0

virtual void set_counter_value (counter_value const&) = 0

virtual bool start () = 0

virtual bool stop () = 0

virtual void reinit (bool reset) = 0

```

```
namespace hpx

namespace performance_counters

class performance_counter_set
```

Public Functions

```
performance_counter_set (bool print_counters_locally = false)
```

Create an empty set of performance counters.

```
performance_counter_set (std::string const &names, bool reset = false)
```

Create a set of performance counters from a name, possibly containing wild-card characters

```
performance_counter_set (std::vector<std::string> const &names, bool reset = false)
```

```
void add_counters (std::string const &names, bool reset = false, error_code &ec = throws)
```

Add more performance counters to the set based on the given name, possibly containing wild-card characters

```
void add_counters (std::vector<std::string> const &names, bool reset = false, error_code &ec = throws)
```

```
std::vector<counter_info> get_counter_infos () const
```

Retrieve the counter infos for all counters in this set.

```
std::vector<hpx::future<counter_value>> get_counter_values (bool reset = false)
```

```
const
```

Retrieve the values for all counters in this set supporting this operation

```
std::vector<counter_value> get_counter_values (launch::sync_policy, bool reset = false,
```

```
error_code &ec = throws) const
```

```
std::vector<hpx::future<counter_values_array>> get_counter_values_array (bool
```

```
reset =
```

```
false)
```

```
const
```

Retrieve the array-values for all counters in this set supporting this operation

```
std::vector<counter_values_array> get_counter_values_array (launch::sync_policy,
```

```
bool reset = false,
```

```
error_code &ec =
```

```
throws) const
```

```
std::vector<hpx::future<void>> reset ()
```

Reset all counters in this set.

```
void reset (launch::sync_policy, error_code &ec = throws)
```

```
std::vector<hpx::future<bool>> start ()
```

Start all counters in this set.

```
bool start (launch::sync_policy, error_code &ec = throws)
```

```
std::vector<hpx::future<bool>> stop ()
```

Stop all counters in this set.

```

bool stop (launch::sync_policy, error_code &ec = throws)
    std::vector<hpx::future<void>> reinit (bool reset = true)
        Re-initialize all counters in this set.

    void reinit (launch::sync_policy, bool reset = true, error_code &ec = throws)
    void release ()
        Release all references to counters in the set.

    std::size_t size () const
        Return the number of counters in this set.

template<typename T>
hpx::future<std::vector<T>> get_values (bool reset = false) const

template<typename T>
std::vector<T> get_values (launch::sync_policy, bool reset = false, error_code &ec =
    throws) const

std::size_t get_invocation_count () const

```

Protected Functions

```
bool find_counter (counter_info const &info, bool reset, error_code &ec)
```

Protected Static Functions

```

template<typename T>
static std::vector<T> extract_values (std::vector<hpx::future<counter_value>>
    &&values)

```

Private Types

```
typedef lcos::local::spinlock mutex_type
```

Private Members

```

mutex_type mtx_
std::vector<counter_info> infos_
std::vector<naming::id_type> ids_
std::vector<std::uint8_t> reset_
std::uint64_t invocation_count_
bool print_counters_locally_

```

```
namespace hpx
```

```
namespace agas
```

Functions

```
void primary_namespace_register_counter_types(error_code &ec = throws)
namespace hpx

namespace util

class query_counters
```

Public Functions

```
query_counters(std::vector<std::string> const &names, std::vector<std::string> const
&reset_names, std::int64_t interval, std::string const &dest, std::string
const &form, std::vector<std::string> const &shortnames, bool
csv_header, bool print_counters_locally, bool counter_types)

~query_counters()

void start()

void stop_evaluating_counters(bool terminate = false)

bool evaluate(bool force = false)

void terminate()

void start_counters(error_code &ec = throws)

void stop_counters(error_code &ec = throws)

void reset_counters(error_code &ec = throws)

void reinit_counters(bool reset = true, error_code &ec = throws)

bool evaluate_counters(bool reset = false, char const *description = nullptr, bool force
= false, error_code &ec = throws)
```

Protected Functions

```
void find_counters()

bool print_raw_counters(bool destination_is_cout, bool reset, bool
no_output, char const *description,
std::vector<performance_counters::counter_info> const
&infos, error_code &ec)

bool print_array_counters(bool destination_is_cout, bool reset, bool
no_output, char const *description,
std::vector<performance_counters::counter_info> const
&infos, error_code &ec)

template<typename Stream>
void print_headers(Stream &output, std::vector<performance_counters::counter_info>
const &infos)
```

```
template<typename Stream, typename Future>
void print_values (Stream *output, std::vector<Future>&&, std::vector<std::size_t> &&indices, std::vector<performance_counters::counter_info> const &infos)

template<typename Stream>
void print_value (Stream *out, performance_counters::counter_info const &infos, performance_counters::counter_value const &value)

template<typename Stream>
void print_value (Stream *out, performance_counters::counter_info const &infos, performance_counters::counter_values_array const &value)

template<typename Stream>
void print_name_csv (Stream &out, std::string const &name)

template<typename Stream>
void print_value_csv (Stream *out, performance_counters::counter_info const &infos, performance_counters::counter_value const &value)

template<typename Stream>
void print_value_csv (Stream *out, performance_counters::counter_info const &infos, performance_counters::counter_values_array const &value)

template<typename Stream>
void print_name_csv_short (Stream &out, std::string const &name)
```

Private Types

```
typedef lcos::local::mutex mutex_type
```

Private Functions

```
query_counters *this_()
```

Private Members

```
mutex_type mtx_
std::vector<std::string> names_
std::vector<std::string> reset_names_
performance_counters::performance_counter_set counters_
std::string destination_
std::string format_
std::vector<std::string> counter_shortnames_
bool csv_header_
bool print_counters_locally_
bool counter_types_
interval_timer timer_
```

```
namespace hpx

namespace performance_counters

class registry
```

Public Functions

```
registry()
```

```
void clear()
```

Reset registry by deleting all stored counter types.

```
counter_status add_counter_type(counter_info const &info, create_counter_func
                                const &create_counter, discover_counters_func
                                const &discover_counters, error_code &ec =
                                throws)
```

Add a new performance counter type to the (local) registry.

```
counter_status discover_counter_types(discover_counter_func discover_counter, discover_counters_mode mode, error_code &ec =
                                         throws)
```

Call the supplied function for all registered counter types.

```
counter_status discover_counter_type(std::string const &fullname, discover_counter_func discover_counter, discover_counters_mode mode, error_code &ec =
                                         throws)
```

Call the supplied function for the given registered counter type.

```
counter_status discover_counter_type(counter_info const &info, discover_counter_func const &f, discover_counters_mode mode, error_code &ec =
                                         throws)
```

```
counter_status get_counter_create_function(counter_info const &info, create_counter_func &create_counter, error_code &ec = throws) const
```

Retrieve the counter creation function which is associated with a given counter type.

```
counter_status get_counter_discovery_function(counter_info const &info, discover_counters_func &func, error_code &ec) const
```

Retrieve the counter discovery function which is associated with a given counter type.

```
counter_status remove_counter_type(counter_info const &info, error_code &ec =
                                         throws)
```

Remove an existing counter type from the (local) registry.

Note This doesn't remove existing counters of this type, it just inhibits defining new counters using this type.

```
counter_status create_raw_counter_value(counter_info const &info, std::int64_t
                                         *countervalue, naming::gid_type &id, error_code &ec = throws)
```

Create a new performance counter instance of type raw_counter based on given counter value.

```

counter_status create_raw_counter(counter_info const &info,
                                hpx::util::function_nonser<std::int64_t>
                                > const &fnaming::gid_type &id, error_code &ec = throwsCreate a new performance counter
instance of type raw_counter based on given function returning the counter value.

counter_status create_raw_counter(counter_info const &info,
                                hpx::util::function_nonser<std::int64_t> bool
                                > const &f, naming::gid_type &id, error_code &ec = throwsCreate a new performance counter
instance of type raw_counter based on given function returning the counter value.

counter_status create_raw_counter(counter_info const &info,
                                hpx::util::function_nonser<std::vector<std::int64_t>>
                                > const &fnaming::gid_type &id, error_code &ec = throwsCreate a new performance counter
instance of type raw_counter based on given function returning the counter value.

counter_status create_raw_counter(counter_info const &info,
                                hpx::util::function_nonser<std::vector<std::int64_t>> bool
                                > const &f, naming::gid_type &id, error_code &ec = throwsCreate a new performance counter
instance of type raw_counter based on given function returning the counter value.

counter_status create_counter(counter_info const &info, naming::gid_type &id, error_code &ec = throws)
Create a new performance counter instance based on given counter info.

counter_status create_statistics_counter(counter_info const &info, std::string
                                         const &base_counter_name,
                                         std::vector<std::size_t> const &parameters,
                                         naming::gid_type &id, error_code
                                         &ec = throws)
Create a new statistics performance counter instance based on given base counter name and given
base time interval (milliseconds).

counter_status create_arithmetics_counter(counter_info const &info,
                                             std::vector<std::string> const
                                             &base_counter_names, naming::gid_type &id, error_code &ec
                                             = throws)
Create a new arithmetics performance counter instance based on given base counter names.

counter_status create_arithmetics_counter_extended(counter_info
                                                    const &info,
                                                    std::vector<std::string>
                                                    const &base_counter_names,
                                                    naming::gid_type &id,
                                                    error_code &ec = throws)
Create a new extended arithmetics performance counter instance based on given base counter
names.

counter_status add_counter(naming::id_type const &id, counter_info const &info, error_code &ec = throws)
Add an existing performance counter instance to the registry.

counter_status remove_counter(counter_info const &info, naming::id_type const &id,
                               error_code &ec = throws)
remove the existing performance counter from the registry

counter_status get_counter_type(std::string const &name, counter_info &info, error_code &ec = throws)
Retrieve counter type information for given counter name.

```

Public Static Functions

```
static registry &instance()
```

Protected Functions

```
counter_type_map_type::iterator locate_counter_type(std::string const &type_name)
counter_type_map_type::const_iterator locate_counter_type(std::string const &type_name) const
```

Private Types

```
typedef std::map<std::string, counter_data> counter_type_map_type
```

Private Members

```
counter_type_map_type countertypes_
struct counter_data
```

Public Functions

```
counter_data(counter_info const &info, create_counter_func const &create_counter, discover_counters_func const &discover_counters)
```

Public Members

```
counter_info info_
create_counter_func create_counter_
discover_counters_func discover_counters_
namespace hpx
```

```
namespace agas
```

Functions

```
void symbol_namespace_register_counter_types(error_code &ec = throws)
namespace hpx
```

```
namespace performance_counters
```

Functions

```
void register_threadmanager_counter_types(threads::threadmanager &tm)

namespace hpx

namespace performance_counters

namespace server

template<typename Operation>
class arithmetics_counter : public hpx::performance_counters::server::base_performance_counter, pub
```

Public Types

```
template<>
using type_holder = arithmetics_counter

template<>
using base_type_holder = base_performance_counter
```

Public Functions

```
arithmetics_counter()

arithmetics_counter(counter_info const &info, std::vector<std::string> const
&base_counter_names)

hpx::performance_counters::counter_value get_counter_value(bool reset = false)
Overloads from the base_counter base class.

bool start()

bool stop()

void reset_counter_value()

void finalize()

naming::address get_current_address() const
```

Private Types

```
template<>
using base_type = components::component_base<arithmetics_counter<Operation>>
```

Private Members

```
    performance_counter_set counters_  
namespace hpx
```

```
    namespace performance_counters  
  
        namespace server
```

```
template<typename Statistic>  
class arithmetics_counter_extended : public hpx::performance_counters::server::base_performance_
```

Public Types

```
template<>  
using type_holder = arithmetics_counter_extended  
  
template<>  
using base_type_holder = base_performance_counter
```

Public Functions

```
arithmetics_counter_extended()  
  
arithmetics_counter_extended(counter_info const &info,  
                           std::vector<std::string> const &base_counter_names)  
  
hpx::performance_counters::counter_value get_counter_value (bool reset = false)  
    Overloads from the base_counter base class.  
  
    bool start ()  
  
    bool stop ()  
  
    void reset_counter_value ()  
  
    void finalize ()  
  
naming::address get_current_address () const
```

Private Types

```
template<>  
using base_type = components::component_base<arithmetics_counter_extended<Statistic>>
```

Private Members

```
performance_counter_set counters_

namespace hpx

namespace performance_counters

namespace server

class base_performance_counter : public hpx::performance_counters::performance_counter_base, pub-
    Subclassed by hpx::performance_counters::server::arithmetics_counter< Operation >, hpx::performance_counters::server::arithmetics_counter_extended< Statistic >, hpx::performance_counters::server::elapsed_time_counter, hpx::performance_counters::server::raw_counter, hpx::performance_counters::server::raw_values_counter, hpx::performance_counters::server::statistics_counter< Statistic >
```

Public Types

```
using wrapping_type = components::component<base_performance_counter>
using base_type_holder = base_performance_counter
```

Public Functions

```
base_performance_counter()
base_performance_counter(counter_info const &info)
constexpr void finalize()
counter_info get_counter_info_nonvirt() const
counter_value get_counter_value_nonvirt(bool reset)
counter_values_array get_counter_values_array_nonvirt(bool reset)
void set_counter_value_nonvirt(counter_value const &info)
void reset_counter_value_nonvirt()
bool start_nonvirt()
bool stop_nonvirt()
void reinit_nonvirt(bool reset)

HPX_DEFINE_COMPONENT_ACTION(base_performance_counter,
                           get_counter_info_nonvirt,
                           get_counter_info_action)
```

The *get_counter_info_action* retrieves a performance counters information. The *get_counter_value_action* queries the value of a performance counter.

```
get_counter_value_action HPX_DEFINE_COMPONENT_ACTION(base_performance_counter,  
                                         get_counter_values_array_nonvirt,  
                                         get_counter_values_array_action)
```

The *set_counter_value_action*.

The *get_counter_value_action* queries the value of a performance counter.

```
get_counter_value_action set_counter_value_action hpx::performance_counters::  
The reset_counter_value_action.
```

The *start_action*

```
get_counter_value_action set_counter_value_action start_action hpx::performan  
The stop_action.
```

The *reinit_action*

Public Members

```
hpx::performance_counters::server::base_performance_counter::get_counter_valu  
get_counter_value_action set_counter_value_nonvirt  
get_counter_value_action set_counter_value_action hpx::performance_counters::  
get_counter_value_action set_counter_value_action start_action hpx::performan
```

Public Static Functions

```
static components::component_type get_component_type ()  
static void set_component_type (components::component_type t)
```

Protected Functions

```
void reset_counter_value ()  
void set_counter_value (counter_value const&)  
counter_value get_counter_value (bool)  
counter_values_array get_counter_values_array (bool)  
bool start ()  
bool stop ()  
void reinit (bool)  
counter_info get_counter_info () const
```

Protected Attributes

```
hpx::performance_counters::counter_info info_  

util::atomic_count invocation_count_  

namespace hpx
```

```
namespace agas
```

Functions

```
naming::gid_type component_namespace_statistics_counter (std::string name) const  

HPX_DEFINE_PLAIN_ACTION (component_namespace_statistics_counter,  

    component_namespace_statistics_counter_action) compo-  

namespace hpx
```

```
namespace performance_counters
```

```
namespace server
```

```
class elapsed_time_counter : public hpx::performance_counters::server::base_performance_counter, pu
```

Public Types

```
using type_holder = elapsed_time_counter  

using base_type_holder = base_performance_counter
```

Public Functions

```
elapsed_time_counter()  

elapsed_time_counter (counter_info const &info)  

hpx::performance_counters::counter_value get_counter_value (bool reset)  

void reset_counter_value()  

bool start()  

bool stop()  

void finalize()  

naming::address get_current_address() const
```

Private Types

```
using base_type = components::component_base<elapsed_time_counter>

namespace hpx
```

```
namespace agas
```

Functions

```
naming::gid_type locality_namespace_statistics_counter(std::string const &name)

HPX_DEFINE_PLAIN_ACTION(locality_namespace_statistics_counter,
                        locality_namespace_statistics_counter_action)

namespace hpx
```

```
namespace agas
```

Functions

```
naming::gid_type primary_namespace_statistics_counter(std::string const &name)

HPX_DEFINE_PLAIN_ACTION(primary_namespace_statistics_counter,
                        primary_namespace_statistics_counter_action)

namespace hpx
```

```
namespace performance_counters
```

```
namespace server
```

```
class raw_counter : public hpx::performance_counters::server::base_performance_counter, public compo
```

Public Types

```
using type_holder = raw_counter

using base_type_holder = base_performance_counter
```

Public Functions

```
raw_counter()

raw_counter(counter_info const &info, hpx::util::function_nonser<std::int64_t> bool
            >f

hpx::performance_counters::counter_value get_counter_value(bool reset = false)

void reset_counter_value()
```

```

void finalize()

naming::address get_current_address () const

Private Types

using base_type = components::component_base<raw_counter>

Private Members

hpx::util::function_nonser<std::int64_t (bool)> f_
bool reset_

namespace hpx

namespace performance_counters

namespace server

class raw_values_counter : public hpx::performance_counters::server::base_performance_counter, public

```

Public Types

```

using type_holder = raw_values_counter
using base_type_holder = base_performance_counter

```

Public Functions

```

raw_values_counter ()

raw_values_counter (counter_info const &info, hpx::util::function_nonser<std::vector<std::int64_t>> f)
    >f

hpx::performance_counters::counter_values_array get_counter_values_array (bool
    reset
    =
    false)

void reset_counter_value ()

void finalize()

naming::address get_current_address () const

```

Private Types

```
using base_type = components::component_base<raw_values_counter>
```

Private Members

```
hpx::util::function_nonsert<std::vector<std::int64_t>bool> hpx::performance_c
bool reset_
```

```
namespace hpx
```

```
namespace performance_counters
```

```
namespace server
```

```
template<typename Statistic>
```

```
class statistics_counter : public hpx::performance_counters::server::base_perfor
```

Public Types

```
typedef statistics_counter type_holder
```

```
typedef base_performance_counter base_type_holder
```

Public Functions

```
statistics_counter()
```

```
statistics_counter(counter_info const &info, std::string const
&base_counter_name, std::size_t parameter1, std::size_t
parameter2, bool reset_base_counter)
```

```
hpx::performance_counters::counter_value get_counter_value(bool reset = false)
Overloads from the base_counter base class.
```

```
bool start()
```

```
bool stop()
```

```
void reset_counter_value()
```

```
void on_terminate()
```

```
void finalize()
```

```
naming::address get_current_address() const
```

Protected Functions

```
bool evaluate_base_counter(counter_value &value)
bool evaluate()
bool ensure_base_counter()
```

Private Types

```
typedef components::component_base<statistics_counter<Statistic>> base_type
typedef lcos::local::spinlock mutex_type
```

Private Functions

```
constexpr statistics_counter *this_()
```

Private Members

```
mutex_type mtx_
hpx::util::interval_timer timer_
std::string base_counter_name_
naming::id_type base_counter_id_
std::unique_ptr<detail::counter_type_from_statistic_base> value_
counter_value prev_value_
bool has_prev_value_
std::size_t parameter1_
std::size_t parameter2_
bool reset_base_counter_
```

namespace hpx

namespace agas

Functions

```
naming::gid_type symbol_namespace_statistics_counter (std::string const &name)
HPX_DEFINE_PLAIN_ACTION (symbol_namespace_statistics_counter, symbol_namespace_statistics_counter_action)
```

plugin_factories

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

HPX_REGISTER_BINARY_FILTER_FACTORY (*BinaryFilter*, *pluginname*)

This macro is used to register a minimal component factory with `Hpx.Plugin`.

```
namespace hpx
```

```
namespace plugins
```

```
template<typename BinaryFilter>
struct binary_filter_factory : public hpx::plugins::binary_filter_factory_base
    #include <binary_filter_factory.hpp> The message_handler_factory provides a minimal implementation of a message handler's factory. If no additional functionality is required this type can be used to implement the full set of minimally required functions to be exposed by a message handler's factory instance.
```

Template Parameters

- `BinaryFilter`: The message handler type this factory should be responsible for.

Public Functions

```
binary_filter_factory(util::section const *global, util::section const *local, bool
                      isenabled)
```

Construct a new factory instance.

Note The contents of both sections has to be cloned in order to save the configuration setting for later use.

Parameters

- `global`: [in] The pointer to a `hpx::util::section` instance referencing the settings read from the [settings] section of the global configuration file (hpx.ini) This pointer may be `nullptr` if no such section has been found.
- `local`: [in] The pointer to a `hpx::util::section` instance referencing the settings read from the section describing this component type: [hpx.components.<name>], where <name> is the instance name of the component as given in the configuration files.

```
~binary_filter_factory()
```

```
serialization::binary_filter *create(bool compress, serialization::binary_filter *next_filter =
                                      nullptr)
```

Create a new instance of a message handler

return Returns the newly created instance of the message handler supported by this factory

Protected Attributes

```
util::section global_settings_
util::section local_settings_
bool isenabled_
```

Defines

HPX_REGISTER_BINARY_FILTER_FACTORY_BASE (*FactoryType, pluginname*)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the component factories.

```
namespace hpx
```

```
namespace plugins
```

```
struct binary_filter_factory_base : public hpx::plugins::plugin_factory_base
#include <binary_filter_factory_base.hpp> The plugin_factory_base has to be used as a base class
for all plugin factories.
```

Subclassed by *hpx::plugins::binary_filter_factory< BinaryFilter >*

Public Functions

```
~binary_filter_factory_base()
```

```
virtual serialization::binary_filter *create(bool compress, serialization::binary_filter
*next_filter = nullptr) = 0
```

Create a new instance of a binary filter

return Returns the newly created instance of the binary filter supported by this factory

```
namespace hpx
```

```
namespace plugins
```

Functions

```
std::vector<parcelport_factory_base*> &get_parcelport_factories()
```

```
void add_parcelport_factory(parcelport_factory_base *factory)
```

```
struct parcelport_factory_base
```

```
#include <parcelport_factory_base.hpp> The plugin_factory_base has to be used as a base class for
all plugin factories.
```

Public Functions

```
virtual ~parcelport_factory_base()

virtual void get_plugin_info(std::vector<std::string> &fillini) = 0

virtual void init(int *argc, char ***argv, util::command_line_handling &cfg) = 0

virtual parcelset::parcelport *create(hpx::util::runtime_configuration const &cfg,
                                      threads::policies::callback_notifier const &notifier) = 0
```

Create a new instance of a parcelport

return Returns the newly created instance of the parcelport supported by this factory

Defines

HPX_REGISTER_PLUGIN_FACTORY_BASE (*FactoryType, pluginname*)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the component factories.

HPX_REGISTER_PLUGIN_MODULE ()

This macro is used to define the required Hpx.Plugin entry points. This macro has to be used in exactly one compilation unit of a component module.

HPX_REGISTER_PLUGIN_MODULE_DYNAMIC ()

template<>

```
struct virtual_constructor<hpx::plugins::plugin_factory_base>
```

Public Types

```
template<>
using type = hpx::util::pack<hpx::util::section const*, hpx::util::section const*, bool>

namespace hpx

namespace plugins

struct plugin_factory_base
#include <plugin_factory_base.hpp> The plugin_factory_base has to be used as a base class for all
plugin factories.

Subclassed by hpx::plugins::binary_filter_factory_base
```

Public Functions

```

virtual ~plugin_factory_base()

namespace util

namespace plugin

template<>
struct virtual_constructor<hpx::plugins::plugin_factory_base>

```

Public Types

```

template<>
using type = hpx::util::pack<hpx::util::section const*, hpx::util::section const*, bool>

```

Defines

HPX_REGISTER_PLUGIN_REGISTRY(...)

This macro is used to create and register a minimal plugin registry with Hpx.Plugin.

HPX_REGISTER_PLUGIN_REGISTRY_(...)

HPX_REGISTER_PLUGIN_REGISTRY_2(PluginType, pluginname)

HPX_REGISTER_PLUGIN_REGISTRY_4(PluginType, pluginname, pluginsection, pluginsuffix)

HPX_REGISTER_PLUGIN_REGISTRY_5(PluginType, pluginname, pluginstring, pluginsection, pluginsuffix)

namespace hpx

namespace plugins

```

template<typename Plugin, char const *const Name, char const *const Section, char const *const Suffix>
struct plugin_registry : public plugin_registry_base

```

#include <plugin_registry.hpp> The plugin_registry provides a minimal implementation of a plugin's registry. If no additional functionality is required this type can be used to implement the full set of minimally required functions to be exposed by a plugin's registry instance.

Template Parameters

- **Plugin:** The plugin type this registry should be responsible for.

Public Functions

```
bool get_plugin_info(std::vector<std::string> &fillini)  
    Return the ini-information for all contained components.
```

Return Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

Parameters

- *fillini*: [in] The module is expected to fill this vector with the ini-information (one line per vector element) for all components implemented in this module.

Defines

```
HPX_DEF_UNIQUE_PLUGIN_NAME(PluginType, name)
```

resiliency_distributed

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace resiliency
```

```
namespace experimental
```

Functions

```
template<typename Action, typename ...Ts>  
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_in
```

```
template<typename Pred, typename Action, typename ...Ts>
```

```
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke
```

```
namespace hpx
```

```
    namespace resiliency
```

```
        namespace experimental
```

Functions

```
template<typename Vote, typename Pred, typename Action, typename ...Ts>
```

```
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke
```

```
template<typename Vote, typename Action, typename ...Ts>
```

```
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke
```

```
template<typename Pred, typename Action, typename ...Ts>
```

```
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke
```

```
template<typename Action, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke
```

runtime_components

See *Public API* for a list of names and headers that are part of the public HPX API.

Defines

HPX_REGISTER_COMPONENT (*type, name, mode*)

Define a component factory for a component type.

This macro is used to create and register a minimal component factory for a component type which allows it to be remotely created using the `hpx::new_<>` function.

This macro can be invoked with one, two or three arguments

Parameters

- *type*: The *type* parameter is a (fully decorated) type of the component type for which a factory should be defined.
- *name*: The *name* parameter specifies the name to use to register the factory. This should uniquely (system-wide) identify the component type. The *name* parameter must conform to the C++ identifier rules (without any namespace). If this parameter is not given, the first parameter is used.
- *mode*: The *mode* parameter has to be one of the defined enumeration values of the enumeration `hpx::components::factory_state_enum`. The default for this parameter is `hpx::components::factory_enabled`.

Defines

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY (...)

This macro is used to register a minimal component registry with Hpx.Plugin.

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_(...)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_2 (*ComponentType*, *componentname*)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_3 (*ComponentType*, *componentname*, *state*)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC (...)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_(...)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_2 (*ComponentType*, *componentname*)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_3 (*ComponentType*, *componentname*,
state)

namespace hpx

namespace components

template<typename Component, factory_state_enum state>

struct component_registry : public component_registry_base

#include <component_registry.hpp> The component_registry provides a minimal implementation of a component's registry. If no additional functionality is required this type can be used to implement the full set of minimally required functions to be exposed by a component's registry instance.

Template Parameters

- **Component**: The component type this registry should be responsible for.

Public Functions

**bool get_component_info (std::vector<std::string> &fillini, std::string const &filepath,
 bool is_static = false)**

Return the ini-information for all contained components.

Return Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

Parameters

- *fillini*: [in] The module is expected to fill this vector with the ini-information (one line per vector element) for all components implemented in this module.

void register_component_type ()

Return the unique identifier of the component type this factory is responsible for.

Return Returns the unique identifier of the component type this factory instance is responsible for. This function throws on any error.

Parameters

- *locality*: [in] The id of the locality this factory is responsible for.
- *agas_client*: [in] The AGAS client to use for component id registration (if needed).

namespace hpx

Functions

```
components::server::runtime_support *get_runtime_support_ptr()  
namespace hpx
```

```
namespace components
```

Functions

```
void console_error_sink(naming::id_type const &dst, std::exception_ptr const &e)  
void console_error_sink(std::exception_ptr const &e)
```

```
namespace hpx
```

```
namespace components
```

Functions

```
void console_logging(logging_destination dest, std::size_t level, std::string const &msg)  
void cleanup_logging()
```

```
void activate_logging()
```

```
namespace hpx
```

```
namespace components
```

Functions

```
template<typename Component, typename ...Ts>  
future<naming::id_type> create_async(naming::id_type const &gid, Ts&&... vs)  
Asynchronously create a new instance of a component.
```

```
template<typename Component, typename ...Ts>  
future<std::vector<naming::id_type>> bulk_create_async(naming::id_type const &gid,  
std::size_t count, Ts&&... vs)
```

```
template<typename Component, typename ...Ts>  
naming::id_type create(naming::id_type const &gid, Ts&&... vs)
```

```
template<typename Component, typename ...Ts>  
std::vector<naming::id_type> bulk_create(naming::id_type const &gid, std::size_t count,  
Ts&&... vs)
```

```
template<typename Component, typename ...Ts>  
future<naming::id_type> create_colocated_async(naming::id_type const &gid, Ts&&...  
vs)
```

```
template<typename Component, typename ...Ts>  
static naming::id_type create_colocated(naming::id_type const &gid, Ts&&... vs)
```

```

template<typename Component, typename ...Ts>
static future<std::vector<naming::id_type>> bulk_create_colocated_async (naming::id_type
    const
    &gid,
    std::size_t
    count,
    Ts&&...
    vs)

template<typename Component, typename ...Ts>
std::vector<naming::id_type> bulk_create_colocated (naming::id_type const &id,
    std::size_t count, Ts&&... vs)

namespace hpx

```

```
namespace components
```

Variables

```
const default_distribution_policy default_layout = {}
```

A predefined instance of the default *distribution_policy*. It will represent the local locality and will place all items to create here.

```
struct default_distribution_policy
```

#include <default_distribution_policy.hpp> This class specifies the parameters for a simple distribution policy to use for creating (and evenly distributing) a given number of items on a given set of localities.

Public Functions

```
constexpr default_distribution_policy()
```

Default-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 package.

Note This function is part of the creation policy implemented by this

`hpx::id_type get_next_target() const`
Returns the locality which is anticipated to be the next target for the action.

struct **async result**

Public Types

```
template<>
using type = hpx::future<typename traits::promise_local_result<typename hpx::traits::extract_action<Ac
```

Defines

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY(...)`

This macro is used to create and register a minimal component factory with `Hpx.Plugin`. This macro may be used if the registered component factory is the only factory to be exposed from a particular module. If more than one factory needs to be exposed the `HPX_REGISTER_COMPONENT_FACTORY` and `HPX_REGISTER_COMPONENT_MODULE` macros should be used instead.

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY_(...)`

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY_3(ComponentType, componentname, basecomponentname)`

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY_4(ComponentType, componentname, basecomponentname, state)`

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC(...)`

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_(...)`

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_3(ComponentType, componentname, basecomponentname)`

`HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_4(ComponentType, componentname, basecomponentname, state)`

Defines

`HPX_DISTRIBUTED_METADATA_DECLARATION(...)`

`HPX_DISTRIBUTED_METADATA_DECLARATION_(...)`

`HPX_DISTRIBUTED_METADATA_DECLARATION_1(config)`

`HPX_DISTRIBUTED_METADATA_DECLARATION_2(config, name)`

`HPX_DISTRIBUTED_METADATA(...)`

`HPX_DISTRIBUTED_METADATA_(...)`

`HPX_DISTRIBUTED_METADATA_1(config)`

`HPX_DISTRIBUTED_METADATA_2(config, name)`

```
namespace hpx
```

```
namespace components
```

```
namespace server
```

```
template<typename ConfigData, typename Derived = detail::this_type>
class distributed_metadata_base : public hpx::components::component_base<std::conditional<std::is_
```

Public Functions

```
distributed_metadata_base()
distributed_metadata_base(ConfigData const &data)
```

ConfigData **get () const**
Retrieve the configuration data.

Private Members

```
ConfigData data_
namespace hpx
```

Functions

template<typename Component, typename... Ts><unspecified> hpx::new_(id_type const & loc)
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 & loc)
Create multiple new instances of the given Component type on the specified locality.

This function creates multiple new instances of the given Component type on the specified locality and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type>> f =
    hpx::new_-<some_component[]>(hpx::find_here(), 10, ...);
hpx::id_type id = f.get();
```

Return The function returns different types depending on its use:

- If the explicit template argument *Component* represents an array of a component type (i.e. `Component[]`, where `traits::is_component<Component>::value` evaluates to true), the function will return an `hpx::future` object instance which holds a `std::vector<hpx::id_type>`, where each of the items in this vector is a global address of one of the newly created components.
- If the explicit template argument *Component* represents an array of a client side object type (i.e. `Component[]`, where `traits::is_client<Component>::value` evaluates to true), the function will return an `hpx::future` object instance which holds a `std::vector<hpx::id_type>`, where each of the items in this vector is a client side instance of the given type, each representing one of the newly created components.

Parameters

- `locality`: [in] The global address of the locality where the new instance should be created on.
- `count`: [in] The number of component instances to create

- **vs:** [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

template<typename Component, typename DistPolicy, typename... Ts><unspecified> hpx::new_<Component>(DistPolicy, Ts...)

Create one or more new instances of the given Component type based on the given distribution policy.

This function creates one or more new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for global address which can be used to reference the new component instance(s).

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
    hpx::new_<some_component>(hpx::default_layout, ...);
hpx::id_type id = f.get();
```

Return The function returns different types depending on its use:

- If the explicit template argument *Component* represents a component type (`traits::is_component<Component>::value` evaluates to true), the function will return an `hpx::future` object instance which can be used to retrieve the global address of the newly created component.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will return a new instance of that type which can be used to refer to the newly created component instance.

Parameters

- **policy:** [in] The distribution policy used to decide where to place the newly created.
- **vs:** [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

template<typename Component, typename DistPolicy, typename... Ts><unspecified> hpx::new_<Component[]>(DistPolicy, Ts...)

Create multiple new instances of the given Component type on the localities as defined by the given distribution policy.

This function creates multiple new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type>> f =
    hpx::new_<some_component[]>(hpx::default_layout, 10, ...);
hpx::id_type id = f.get();
```

Return The function returns different types depending on its use:

- If the explicit template argument *Component* represents an array of a component type (i.e. `Component[]`, where `traits::is_component<Component>::value` evaluates to true), the function will return an `hpx::future` object instance which holds a `std::vector<hpx::id_type>`, where each of the items in this vector is a global address of one of the newly created components.

- If the explicit template argument *Component* represents an array of a client side object type (i.e. *Component*[], where `traits::is_client<Component>::value` evaluates to true), the function will return an `hpx::future` object instance which holds a `std::vector<hpx::id_type>`, where each of the items in this vector is a client side instance of the given type, each representing one of the newly created components.

Parameters

- `policy`: [in] The distribution policy used to decide where to place the newly created.
- `count`: [in] The number of component instances to create
- `vs`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

```
namespace hpx
```

```
namespace components
```

```
namespace server
```

Functions

```
void console_error_sink(std::exception_ptr const&)
```

```
HPX_DEFINE_PLAIN_ACTION(console_error_sink, console_error_sink_action)
```

```
namespace hpx
```

```
namespace components
```

```
namespace server
```

Functions

```
console_error_dispatcher &get_error_dispatcher()
```

```
class console_error_dispatcher
```

Public Types

```
typedef util::spinlock mutex_type
```

```
typedef util::function_nonser<void (std::string const&)> sink_type
```

Public Functions

```
HPX_NON_COPYABLE(console_error_dispatcher)

console_error_dispatcher()

template<typename F>
sink_type set_error_sink(F &&sink)

void operator() (std::string const &msg)
```

Private Members

```
mutex_type mtx_
sink_type sink_
```

```
namespace hpx
```

```
namespace components
```

TypeDefs

```
typedef hpx::tuple<logging_destination, std::size_t, std::string> message_type
typedef std::vector<message_type> messages_type
```

```
namespace server
```

Functions

```
void console_logging(messages_type const&)

template<typename Dummy = void>
class console_logging_action : public actions::direct_action<void (*) (messages_type
const&),
con-
sole_logging,
con-
sole_logging_action<void>>
```

Public Functions

```
console_logging_action()

console_logging_action(messages_type const &msgs)

console_logging_action(threads::thread_priority, messages_type const &msgs)
```

Public Static Functions

```
template<typename T>
static util::unused_type execute_function(naming::address_type, naming::component_type, T &&v)
```

Private Types

```
typedef actions::direct_action<void (*) (messages_type const&), console_logging, console_logging_action> base_type
```

runtime_distributed

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

```
class runtime_distributed : public runtime
#include <runtime_distributed.hpp> The runtime class encapsulates the HPX runtime system in a simple to use way. It makes sure all required parts of the HPX runtime system are properly initialized.
```

Public Functions

```
hpx::runtime_distributed::runtime_distributed(util::runtime_configuration & rtcfg,
Construct a new HPX runtime instance
```

Parameters

- `locality_mode`: [in] This is the mode the given runtime instance should be executed in.

~runtime_distributed()

The destructor makes sure all HPX runtime services are properly shut down before exiting.

```
int start (util::function_nonser<hpx_main_function_type> const &func, bool blocking = false)
Start the runtime system.
```

Return If a *blocking* is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter `func`. Otherwise it will return zero.

Parameters

- `func`: [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the `typedef hpx_main_function_type`.
- `blocking`: [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function `runtime::start` will call `runtime::wait` internally.

```
int start (bool blocking = false)
Start the runtime system.
```

Return If a *blocking* is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter `func`. Otherwise it will return zero.

Parameters

- **blocking**: [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function `runtime::start` will call `runtime::wait` internally .

```
int wait ()
```

Wait for the shutdown action to be executed.

Return This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

```
void stop (bool blocking = true)
```

Initiate termination of the runtime system.

Parameters

- **blocking**: [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

```
int finalize (double shutdown_timeout)
```

```
void stop_helper (bool blocking, std::condition_variable &cond, std::mutex &mtx)
```

Stop the runtime system, wait for termination.

Parameters

- **blocking**: [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

```
int suspend ()
```

Suspend the runtime system.

```
int resume ()
```

Resume the runtime system.

```
bool report_error (std::size_t num_thread, std::exception_ptr const &e, bool terminate_all = true)
```

Report a non-recoverable error to the runtime system.

Parameters

- **num_thread**: [in] The number of the operating system thread the error has been detected in.
- **e**: [in] This is an instance encapsulating an exception which lead to this function call.
- **terminate_all**: [in] Kill all localities attached to the currently running application (default: *true*)

```
bool report_error (std::exception_ptr const &e, bool terminate_all = true)
```

Report a non-recoverable error to the runtime system.

Note This function will retrieve the number of the current shepherd thread and forward to the `report_error` function above.

Parameters

- **e**: [in] This is an instance encapsulating an exception which lead to this function call.

- `terminate_all`: [in] Kill all localities attached to the currently running application (default: true)

`int run (util::function_nonser<hpx_main_function_type> const &func)`

Run the HPX runtime system, use the given function for the main *thread* and block waiting for all threads to finish.

Note The parameter `func` is optional. If no function is supplied, the runtime system will simply wait for the shutdown action without explicitly executing any main thread.

Return This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

Parameters

- `func`: [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the typedef `hpx_main_function_type`. This parameter is optional and defaults to none main thread function, in which case all threads have to be scheduled explicitly.

`int run ()`

Run the HPX runtime system, initially use the given number of (OS) threads in the thread-manager and block waiting for all threads to finish.

Return This function will always return 0 (zero).

`bool is_networking_enabled ()`

`template<typename F>`

`components::server::console_error_dispatcher::sink_type set_error_sink (F &&sink)`

`performance_counters::registry &get_counter_registry ()`

Allow access to the registry counter registry instance used by the HPX runtime.

`performance_counters::registry const &get_counter_registry () const`

Allow access to the registry counter registry instance used by the HPX runtime.

`void register_counter_types ()`

Install all performance counters related to this runtime instance.

`void register_query_counters (std::shared_ptr<util::query_counters> const &active_counters)`

`void start_active_counters (error_code &ec = throws)`

`void stop_active_counters (error_code &ec = throws)`

`void reset_active_counters (error_code &ec = throws)`

`void reinit_active_counters (bool reset = true, error_code &ec = throws)`

`void evaluate_active_counters (bool reset = false, char const *description = nullptr, error_code &ec = throws)`

`void stop_evaluating_counters (bool terminate = false)`

`naming::resolver_client &get_agas_client ()`

Allow access to the AGAS client instance used by the HPX runtime.

`hpx::threads::threadmanager &get_thread_manager()`

Allow access to the thread manager instance used by the HPX runtime.

`applier::applier &get_applier()`

Allow access to the applier instance used by the HPX runtime.

`std::string here() const`

Returns a string of the locality endpoints (usable in debug output)

`naming::address_type get_runtime_support_lva() const`

`naming::gid_type get_next_id(std::size_t count = 1)`

`void init_id_pool_range()`

`util::unique_id_ranges &get_id_pool()`

`void initialize_agas()`

Initialize AGAS operation.

`void add_pre_startup_function(startup_function_type f)`

Add a function to be executed inside a HPX thread before hpx_main but guaranteed to be executed before any startup function registered with `add_startup_function`.

Note The difference to a startup function is that all pre-startup functions will be (system-wide) executed before any startup function.

Parameters

- `f`: The function ‘`f`’ will be called from inside a HPX thread before `hpx_main` is executed. This is very useful to setup the runtime environment of the application (install performance counters, etc.)

`void add_startup_function(startup_function_type f)`

Add a function to be executed inside a HPX thread before `hpx_main`

Parameters

- `f`: The function ‘`f`’ will be called from inside a HPX thread before `hpx_main` is executed. This is very useful to setup the runtime environment of the application (install performance counters, etc.)

`void add_pre_shutdown_function(shutdown_function_type f)`

Add a function to be executed inside a HPX thread during `hpx::finalize`, but guaranteed before any of the shutdown functions is executed.

Note The difference to a shutdown function is that all pre-shutdown functions will be (system-wide) executed before any shutdown function.

Parameters

- `f`: The function ‘`f`’ will be called from inside a HPX thread while `hpx::finalize` is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

`void add_shutdown_function(shutdown_function_type f)`

Add a function to be executed inside a HPX thread during `hpx::finalize`

Parameters

- `f`: The function ‘`f`’ will be called from inside a HPX thread while `hpx::finalize` is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

```
hpx::util::io_service_pool *get_thread_pool (char const *name)
```

Access one of the internal thread pools (io_service instances) HPX is using to perform specific tasks. The three possible values for the argument `name` are “main_pool”, “io_pool”, “parcel_pool”, and “timer_pool”. For any other argument value the function will return zero.

```
bool register_thread (char const *name, std::size_t num = 0, bool service_thread = true,
                      error_code &ec = throws)
```

Register an external OS-thread with HPX.

```
notification_policy_type get_notification_policy (char const *prefix, run-
                                                time_local::os_thread_type type)
```

Generate a new notification policy instance for the given thread name prefix

```
std::uint32_t get_locality_id (error_code &ec) const
```

```
std::size_t get_num_worker_threads () const
```

```
std::uint32_t get_num_localities (hpx::launch::sync_policy, error_code &ec) const
```

```
std::uint32_t get_initial_num_localities () const
```

```
hpx::future<std::uint32_t> get_num_localities () const
```

```
std::string get_locality_name () const
```

```
std::uint32_t get_num_localities (hpx::launch::sync_policy, components::component_type
                                 type, error_code &ec) const
```

```
hpx::future<std::uint32_t> get_num_localities (components::component_type type) const
```

```
std::uint32_t assign_cores (std::string const &locality_basename, std::uint32_t num_threads)
```

```
std::uint32_t assign_cores ()
```

Private Types

```
using used_cores_map_type = std::map<std::string, std::uint32_t>
```

Private Functions

```
threads::thread_result_type run_helper (util::function_nonser<runtime::hpx_main_function_type>
                                         const &func, int &result)
```

```
void init_global_data ()
```

```
void deinit_global_data ()
```

```
void wait_helper (std::mutex &mtx, std::condition_variable &cond, bool &running)
```

```
void init_tss_helper (char const *context, runtime_local::os_thread_type type, std::size_t
                      local_thread_num, std::size_t global_thread_num, char const
                      *pool_name, char const *postfix, bool service_thread)
```

```
void deinit_tss_helper (char const *context, std::size_t num)
```

```
void init_tss_ex(std::string const &locality, char const *context, runtime_local::os_thread_type type, std::size_t local_thread_num, std::size_t global_thread_num, char const *pool_name, char const *postfix, bool service_thread, error_code &ec)
```

Private Members

```
runtime_mode mode_
util::unique_id_ranges id_pool_
naming::resolver_client agas_client_
applier::applier applier_
used_cores_map_type used_cores_map_
std::unique_ptr<components::server::runtime_support> runtime_support_
std::shared_ptr<util::query_counters> active_counters_
int (*pre_main_)(runtime_mode)
void (*post_main_)()
```

Private Static Functions

```
static void default_errorsink(std::string const&
namespace hpx

namespace applier

class applier
#include <applier.hpp> The applier class is used to decide whether a particular action has to be issued on a local or a remote resource. If the target component is local a new thread will be created, if the target is remote a parcel will be sent.
```

Public Functions

```
HPX_NON_COPYABLE (applier)

applier ()

void init (threads::threadmanager &tm)

~applier ()

void initialize (std::uint64_t rts)

threads::threadmanager &get_thread_manager ()
Access the thread-manager instance associated with this applier.
This function returns a reference to the thread manager this applier instance has been created with.
```

`naming::gid_type const &get_raw_locality(error_code &ec = throws) const`
Allow access to the locality of the locality this applier instance is associated with.

This function returns a reference to the locality this applier instance is associated with.

`std::uint32_t get_locality_id(error_code &ec = throws) const`
Allow access to the id of the locality this applier instance is associated with.

This function returns a reference to the id of the locality this applier instance is associated with.

`bool get_raw_remote_localities(std::vector<naming::gid_type> &locality_ids,
components::component_type type = components::component_invalid, error_code &ec =
throws) const`

Return list of localities of all remote localities registered with the AGAS service for a specific component type.

This function returns a list of all remote localities (all localities known to AGAS except the local one) supporting the given component type.

Return The function returns *true* if there is at least one remote locality known to the AGASService (!prefixes.empty()).

Parameters

- `locality_ids`: [out] The reference to a vector of id_types filled by the function.
- `type`: [in] The type of the component which needs to exist on the returned localities.

`bool get_remote_localities(std::vector<naming::id_type> &locality_ids,
components::component_type type = components::component_invalid, error_code &ec = throws)
const`

`bool get_raw_localities(std::vector<naming::gid_type> &locality_ids,
components::component_type type = components::component_invalid) const`

Return list of locality_ids of all localities registered with the AGAS service for a specific component type.

This function returns a list of all localities (all localities known to AGAS except the local one) supporting the given component type.

Return The function returns *true* if there is at least one remote locality known to the AGASService (!prefixes.empty()).

Parameters

- `locality_ids`: [out] The reference to a vector of id_types filled by the function.
- `type`: [in] The type of the component which needs to exist on the returned localities.

`bool get_localities(std::vector<naming::id_type> &locality_ids, error_code &ec =
throws) const`

`bool get_localities(std::vector<naming::id_type> &locality_ids, components::component_type type, error_code &ec = throws) const`

`naming::gid_type const &get_runtime_support_raw_gid() const`

By convention the runtime_support has a gid identical to the prefix of the locality the runtime_support is responsible for

`naming::id_type const &get_runtime_support_gid() const`

By convention the runtime_support has a gid identical to the prefix of the locality the runtime_support is responsible for

Private Members

```
threads::threadmanager *thread_manager_
naming::id_type runtime_support_id_
namespace hpx
```

```
namespace applier
```

Functions

```
applier &get_applier()
```

The function *get_applier* returns a reference to the (thread specific) applier instance.

```
applier *get_applier_ptr()
```

The function *get_applier* returns a pointer to the (thread specific) applier instance. The returned pointer is NULL if the current thread is not known to HPX or if the runtime system is not active.

```
namespace applier
```

The namespace *applier* contains all definitions needed for the class `hpx::applier::applier` and its related functionality. This namespace is part of the HPX core module.

```
namespace hpx
```

```
namespace components
```

Functions

```
template<typename Component>
```

```
future<naming::id_type> copy(naming::id_type const &to_copy)
```

Copy given component to the specified target locality.

The function *copy*<*Component*> will create a copy of the component referenced by *to_copy* on the locality specified with *target_locality*. It returns a future referring to the newly created component instance.

Return A future representing the global id of the newly (copied) component instance.

Note The new component instance is created on the locality of the component instance which is to be copied.

Parameters

- *to_copy*: [in] The global id of the component to copy

Template Parameters

- The: only template argument specifies the component type to create.

```
template<typename Component>
```

```
future<naming::id_type> copy(naming::id_type const &to_copy, naming::id_type const &target_locality)
```

Copy given component to the specified target locality.

The function *copy*<*Component*> will create a copy of the component referenced by *to_copy* on the locality specified with *target_locality*. It returns a future referring to the newly created component instance.

Return A future representing the global id of the newly (copied) component instance.

Parameters

- `to_copy`: [in] The global id of the component to copy
- `target_locality`: [in] The locality where the copy should be created.

Template Parameters

- The: only template argument specifies the component type to create.

```
template<typename Derived, typename Stub>
Derived copy(client_base<Derived, Stub> const &to_copy, naming::id_type const &target_locality = naming::invalid_id)
```

Copy given component to the specified target locality.

The function `copy` will create a copy of the component referenced by the client side object `to_copy` on the locality specified with `target_locality`. It returns a new client side object future referring to the newly created component instance.

Return A future representing the global id of the newly (copied) component instance.

Note If the second argument is omitted (or is `invalid_id`) the new component instance is created on the locality of the component instance which is to be copied.

Parameters

- `to_copy`: [in] The client side object representing the component to copy
- `target_locality`: [in, optional] The locality where the copy should be created (default is same locality as source).

Template Parameters

- The: only template argument specifies the component type to create.

```
namespace hpx
```

Functions

```
naming::id_type find_root_locality(error_code &ec = throws)
```

Return the global id representing the root locality.

The function `find_root_locality()` can be used to retrieve the global id usable to refer to the root locality. The root locality is the locality where the main AGAS service is hosted.

Note Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Return The global id representing the root locality for this application.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note This function will return meaningful results only if called from an HPX-thread. It will return `hpx::naming::invalid_id` otherwise.

See `hpx::find_all_localities()`, `hpx::find_locality()`

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

```
std::vector<naming::id_type> find_all_localities(error_code &ec = throws)
```

Return the list of global ids representing all localities available to this application.

The function `find_all_localities()` can be used to retrieve the global ids of all localities currently available to this application.

Note Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Return The global ids representing the localities currently available to this application.

Note As long as `ec` is not pre-initialized to `hpx::throws` this function doesn't throw but returns the result code using the parameter `ec`. Otherwise it throws an instance of `hpx::exception`.

Note This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

See `hpx::find_here()`, `hpx::find_locality()`

Parameters

- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`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.

`namespace hpx`

Functions

`naming::id_type find_here(error_code &ec = throws)`

Return the global id representing this locality.

The function `find_here()` can be used to retrieve the global id usable to refer to the current locality.

Note Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Return The global id representing the locality this function has been called on.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note This function will return meaningful results only if called from an HPX-thread. It will return *hpx::naming::invalid_id* otherwise.

See `hpx::find_all_localities()`, `hpx::find_locality()`

Parameters

- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

namespace hpx

Functions

std::vector<naming::id_type> find_all_localities(components::component_type type, error_code &ec = throws)

Return the list of global ids representing all localities available to this application which support the given component type.

The function `find_all_localities()` can be used to retrieve the global ids of all localities currently available to this application which support the creation of instances of the given component type.

Note Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Return The global ids representing the localities currently available to this application which support the creation of instances of the given component type. If no localities supporting the given component type are currently available, this function will return an empty vector.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of *hpx::exception*.

Note This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

See `hpx::find_here()`, `hpx::find_locality()`

Parameters

- *type*: [in] The type of the components for which the function should return the available localities.
- *ec*: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

std::vector<naming::id_type> find_remote_localities(components::component_type type, error_code &ec = throws)

Return the list of locality ids of remote localities supporting the given component type. By default this function will return the list of all remote localities (all but the current locality).

The function `find_remote_localities()` can be used to retrieve the global ids of all remote localities currently available to this application (i.e. all localities except the current one) which support the creation of instances of the given component type.

Note Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Return The global ids representing the remote localities currently available to this application.

Note As long as *ec* is not pre-initialized to *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.

namespace hpx

Functions

`future<std::string> get_locality_name (naming::id_type const &id)`

Return the name of the referenced locality.

This function returns a future referring to the name for the locality of the given id.

Return This function returns the name for the locality of the given id. The name is retrieved from the underlying networking layer and may be different for different parcel ports.

See `std::string get_locality_name()`

Parameters

- `id`: [in] The global id of the locality for which the name should be retrieved

`namespace hpx`

Functions

`hpx::future<std::uint32_t> get_num_localities (components::component_type t)`

Asynchronously return the number of localities which are currently registered for the running application.

The function `get_num_localities` asynchronously returns the number of localities currently connected to the console which support the creation of the given component type. The returned future represents the actual result.

Note This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

See `hpx::find_all_localities, hpx::get_num_localities`

Parameters

- `t`: The component type for which the number of connected localities should be retrieved.

`std::uint32_t get_num_localities (launch::sync_policy, components::component_type t, error_code &ec = throws)`

Synchronously return the number of localities which are currently registered for the running application.

The function `get_num_localities` returns the number of localities currently connected to the console which support the creation of the given component type. The returned future represents the actual result.

Note This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

See `hpx::find_all_localities, hpx::get_num_localities`

Parameters

- `t`: The component type for which the number of connected localities should be retrieved.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`namespace hpx`

namespace components**Functions**

```
template<typename Component, typename DistPolicy>
future<naming::id_type> migrate(naming::id_type const &to_migrate, DistPolicy const &policy)
```

Migrate the given component to the specified target locality

The function *migrate*<Component> will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Return A future representing the global id of the migrated component instance. This should be the same as *migrate_to*.

Parameters

- *to_migrate*: [in] The client side representation of the component to migrate.
- *policy*: [in] A distribution policy which will be used to determine the locality to migrate this object to.

Template Parameters

- *Component*: Specifies the component type of the component to migrate.
- *DistPolicy*: Specifies the distribution policy to use to determine the destination locality.

```
template<typename Derived, typename Stub, typename DistPolicy>
Derived migrate(client_base<Derived, Stub> const &to_migrate, DistPolicy const &policy)
```

Migrate the given component to the specified target locality

The function *migrate*<Component> will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Return A future representing the global id of the migrated component instance. This should be the same as *migrate_to*.

Parameters

- *to_migrate*: [in] The client side representation of the component to migrate.
- *policy*: [in] A distribution policy which will be used to determine the locality to migrate this object to.

Template Parameters

- *Derived*: Specifies the component type of the component to migrate.
- *DistPolicy*: Specifies the distribution policy to use to determine the destination locality.

```
template<typename Component>
future<naming::id_type> migrate(naming::id_type const &to_migrate, naming::id_type const &target_locality)
```

Migrate the component with the given id to the specified target locality

The function *migrate*<Component> will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Return A future representing the global id of the migrated component instance. This should be the same as *migrate_to*.

Parameters

- *to_migrate*: [in] The global id of the component to migrate.
- *target_locality*: [in] The locality where the component should be migrated to.

Template Parameters

- *Component*: Specifies the component type of the component to migrate.

```
template<typename Derived, typename Stub>
Derived migrate(client_base<Derived, Stub> const &to_migrate, naming::id_type const
&target_locality)
Migrate the given component to the specified target locality
```

The function *migrate*<*Component*> will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Return A client side representation of representing of the migrated component instance. This should be the same as *migrate_to*.

Parameters

- *to_migrate*: [in] The client side representation of the component to migrate.
- *target_locality*: [in] The id of the locality to migrate this object to.

Template Parameters

- *Derived*: Specifies the component type of the component to migrate.

```
namespace hpx
```

```
namespace components
```

```
class runtime_support : public hpx::components::stubs::runtime_support
#include <runtime_support.hpp> The runtime_support class is the client side representation of a
server::runtime_support component
```

Public Functions

```
runtime_support(naming::id_type const &gid = naming::invalid_id)
```

Create a client side representation for the existing server::runtime_support instance with the given global id *gid*.

```
template<typename Component, typename ...Ts>
naming::id_type create_component(Ts&&... vs)
```

Create a new component type using the *runtime_support*.

```
template<typename Component, typename ...Ts>
```

```
hpx::future<naming::id_type> create_component_async(Ts&&... vs)
```

Asynchronously create a new component using the *runtime_support*.

```
template<typename Component, typename ...Ts>
```

```
std::vector<naming::id_type> bulk_create_component(std::size_t, Ts&&... vs)
```

Asynchronously create N new default constructed components using the *runtime_support*

```
template<typename Component, typename ...Ts>
```

```
hpx::future<std::vector<naming::id_type>> bulk_create_components_async(std::size_t,
Ts&&...
vs)
```

Asynchronously create a new component using the *runtime_support*.

```
hpx::future<int> load_components_async()
```

```
int load_components()
```

```
hpx::future<void> call_startup_functions_async(bool pre_startup)
```

```
void call_startup_functions(bool pre_startup)
```

```
hpx::future<void> shutdown_async (double timeout = -1)
    Shutdown the given runtime system.

void shutdown (double timeout = -1)

void shutdown_all (double timeout = -1)
    Shutdown the runtime systems of all localities.

hpx::future<void> terminate_async ()
    Terminate the given runtime system.

void terminate ()

void terminate_all ()
    Terminate the runtime systems of all localities.

void get_config (util::section &ini)
    Retrieve configuration information.

naming::id_type const &get_id() const
naming::gid_type const &get_raw_gid() const
```

Private Types

```
typedef stubs::runtime_support base_type
```

Private Members

```
naming::id_type gid_
```

```
namespace hpx
```

```
namespace components
```

```
namespace server
```

Functions

```
template<typename Component>
future<naming::id_type> copy_component_here (naming::id_type const &to_copy)

template<typename Component>
future<naming::id_type> copy_component (naming::id_type const &to_copy, naming::id_type const &target_locality)

namespace hpx

namespace components

namespace server
```

Functions

```

template<typename Component, typename DistPolicy>
future<id_type> migrate_component (id_type const &to_migrate, naming::address
                                         const &addr, DistPolicy const &policy)

template<typename Component, typename DistPolicy>
future<id_type> trigger_migrate_component (id_type const &to_migrate, DistPol-
                                         icy const &policy, naming::id_type
                                         const &id, naming::address const
                                         &addr)

template<typename Component, typename DistPolicy>
future<id_type> perform_migrate_component (id_type const &to_migrate, DistPolicy
                                         const &policy)

namespace hpx

namespace components

namespace server

class runtime_support

```

Public Types

typedef *runtime_support* **type_holder**

Public Functions

```

runtime_support (hpx::util::runtime_configuration &cfg)
~runtime_support ()

void delete_function_lists ()

void tidy ()

template<typename Component>
naming::gid_type create_component ()
    Actions to create new objects.

template<typename Component, typename T, typename ...Ts>
naming::gid_type create_component (T v, Ts... vs)

template<typename Component>
std::vector<naming::gid_type> bulk_create_component (std::size_t count)

template<typename Component, typename T, typename ...Ts>
std::vector<naming::gid_type> bulk_create_component (std::size_t count, T v, Ts...
                                         vs)

template<typename Component>
```

```
naming::gid_type copy_create_component (std::shared_ptr<Component> const &p,  
                                         bool local_op)  
  
template<typename Component>  
naming::gid_type migrate_component_to_here (std::shared_ptr<Component>  
                                              const &p,      naming::id_type  
                                              to_migrate)  
  
void shutdown (double timeout, naming::id_type const &respond_to)  
    Gracefully shutdown this runtime system instance.  
  
void shutdown_all (double timeout)  
    Gracefully shutdown runtime system instances on all localities.  
  
HPX_NORETURN void hpx::components::server::runtime_support::terminate(naming::id_type id)  
    Shutdown this runtime system instance.  
  
void terminate_act (naming::id_type const &id)  
  
HPX_NORETURN void hpx::components::server::runtime_support::terminate_all()  
    Shutdown runtime system instances on all localities.  
  
void terminate_all_act ()  
  
util::section get_config()  
    Retrieve configuration information.  
  
int load_components ()  
    Load all components on this locality.  
  
void call_startup_functions (bool pre_startup)  
  
void call_shutdown_functions (bool pre_shutdown)  
  
void garbage_collect ()  
    Force a garbage collection operation in the AGAS layer.  
  
naming::gid_type create_performance_counter (performance_counters::counter_info  
                                              const &info)  
    Create the given performance counter instance.  
  
void remove_from_connection_cache (naming::gid_type      const      &gid,  
                                 parcelset::endpoints_type const &eps)  
    Remove the given locality from our connection cache.  
  
HPX_DEFINE_COMPONENT_ACTION (runtime_support, terminate_act, terminate_action)  
    termination detection  
  
terminate_all_action HPX_DEFINE_COMPONENT_ACTION (runtime_support,           re-  
                                                 move_from_connection_cache)  
    Start the runtime_support component.  
  
void wait ()  
    Wait for the runtime_support component to notify the calling thread.
```

This function will be called from the main thread, causing it to block while the HPX functionality is executed. The main thread will block until the *shutdown_action* is executed, which in turn notifies all waiting threads.

```
void stop(double timeout, naming::id_type const &respond_to, bool re-  

move_from_remote_caches)  

Notify all waiting (blocking) threads allowing the system to be properly stopped.
```

Note This function can be called from any thread.

```
void stopped()  

    called locally only  

void notify_waiting_main()  

bool was_stopped() const  

void add_pre_startup_function(startup_function_type f)  

void add_startup_function(startup_function_type f)  

void add_pre_shutdown_function(shutdown_function_type f)  

void add_shutdown_function(shutdown_function_type f)  

void remove_here_from_connection_cache()  

void remove_here_from_console_connection_cache()
```

Public Members

```
hpx::components::server::runtime_support::terminate_all_act
```

Public Static Functions

```
static component_type get_component_type()  

static void set_component_type(component_type t)  

static constexpr void finalize()  

    finalize() will be called just before the instance gets destructed
```

Parameters

- *self*: [in] The HPX *thread* used to execute this function.
- *appl*: [in] The applier to be used for finalization of the component instance.

```
static bool is_target_valid(naming::id_type const &id)
```

Protected Functions

```
int load_components(util::section &ini, naming::gid_type const &prefix,  

naming::resolver_client &agas_client, hpx::program_options::options_description &options,  

std::set<std::string> &startup_handled)
```

```
bool load_component (hpx::util::plugin::dll &d, util::section &ini, std::string const
&instance, std::string const &component, filesystem::path
const &lib, naming::gid_type const &prefix, naming::resolver_client &agas_client, bool isdefault, bool isenabled,
hpx::program_options::options_description &options,
std::set<std::string> &startup_handled)

bool load_component_dynamic (util::section &ini, std::string const &instance,
std::string const &component, filesystem::path
lib, naming::gid_type const &prefix, naming::resolver_client &agas_client, bool isdefault, bool
isenabled, hpx::program_options::options_description
&options, std::set<std::string> &startup_handled)

bool load_startup_shutdown_functions (hpx::util::plugin::dll &d, error_code
&ec)

bool load_commandline_options (hpx::util::plugin::dll &d,
hpx::program_options::options_description &options, error_code &ec)

bool load_component_static (util::section &ini, std::string const &instance,
std::string const &component, filesystem::path
const &lib, naming::gid_type const &prefix, naming::resolver_client &agas_client, bool isdefault, bool
isenabled, hpx::program_options::options_description
&options, std::set<std::string> &startup_handled)

bool load_startup_shutdown_functions_static (std::string const &mod, error_code &ec)

bool load_commandline_options_static (std::string const &mod,
hpx::program_options::options_description &options, error_code &ec)

bool load_plugins (util::section &ini, hpx::program_options::options_description &options,
std::set<std::string> &startup_handled)

bool load_plugin (hpx::util::plugin::dll &d, util::section &ini, std::string const &instance,
std::string const &component, filesystem::path const &lib,
bool isenabled, hpx::program_options::options_description &options,
std::set<std::string> &startup_handled)

bool load_plugin_dynamic (util::section &ini, std::string const &instance, std::string
const &component, filesystem::path lib, bool isenabled,
hpx::program_options::options_description &options,
std::set<std::string> &startup_handled)

std::size_t dijkstra_termination_detection (std::vector<naming::id_type>
const &locality_ids)
```

Private Types

```
typedef lcos::local::spinlock plugin_map_mutex_type
typedef plugin_factory plugin_factory_type
typedef std::map<std::string, plugin_factory_type> plugin_map_type
typedef std::map<std::string, hpx::util::plugin::dll> modules_map_type
typedef std::vector<static_factory_load_data_type> static_modules_type
```

Private Members

```
std::mutex mtx_
std::condition_variable wait_condition_
std::condition_variable stop_condition_
bool stop_called_
bool stop_done_
bool terminated_
std::thread::id main_thread_id_
std::atomic<bool> shutdown_all_invoked_
plugin_map_mutex_type p_mtx_
plugin_map_type plugins_
modules_map_type &modules_
static_modules_type static_modules_
lcos::local::spinlock globals_mtx_
std::list<startup_function_type> pre_startup_functions_
std::list<startup_function_type> startup_functions_
std::list<shutdown_function_type> pre_shutdown_functions_
std::list<shutdown_function_type> shutdown_functions_

struct plugin_factory
```

Public Functions

```
plugin_factory (std::shared_ptr<plugins::plugin_factory_base> const &f,
                  hpx::util::plugin::dll const &d, bool enabled)
```

Public Members

```
std::shared_ptr<plugins::plugin_factory_base> first
hpx::util::plugin::dll const &second
bool isenabled

namespace hpx

namespace components

namespace stubs

struct runtime_support
Subclassed by hpx::components::runtime_support
```

Public Static Functions

```
template<typename Component, typename ...Ts>
static hpx::future<naming::id_type> create_component_async(naming::id_type
                                                               const &gid,
                                                               Ts&&... vs)
Create a new component type using the runtime_support with the given targetgid. This is a
non-blocking call. The caller needs to call future::get on the result of this function to obtain
the global id of the newly created object.

template<typename Component, typename ...Ts>
static naming::id_type create_component(naming::id_type const &gid, Ts&&...
                                         vs)
Create a new component type using the runtime_support with the given targetgid. Block for
the creation to finish.

template<typename Component, typename ...Ts>
static hpx::future<std::vector<naming::id_type>> bulk_create_component_colocated_async(nam
                                                               const &gid,
                                                               std::size_t
                                                               count,
                                                               Ts&&...
                                                               vs)
Create multiple new components type using the runtime_support colocated with the with the
given targetgid. This is a non-blocking call.

template<typename Component, typename ...Ts>
static std::vector<naming::id_type> bulk_create_component_colocated(naming::id_type
                                                               const &gid,
                                                               std::size_t
                                                               count,
                                                               Ts&&...
                                                               vs)
Create multiple new components type using the runtime_support colocated with the with the
given targetgid. Block for the creation to finish.
```

```
template<typename Component, typename ...Ts>
static hpx::future<std::vector<naming::id_type>> bulk_create_component_async (naming::id_type
const
&gid,
std::size_t count,
Ts&&...
vs)
```

Create multiple new components *type* using the *runtime_support* on the given locality. This is a non-blocking call.

```
template<typename Component, typename ...Ts>
static std::vector<naming::id_type> bulk_create_component (naming::id_type
const &gid,
std::size_t count,
Ts&&... vs)
```

Create multiple new components *type* using the *runtime_support* on the given locality. Block for the creation to finish.

```
template<typename Component, typename ...Ts>
static hpx::future<naming::id_type> create_component_colocated_async (naming::id_type
const
&gid,
Ts&&...
vs)
```

Create a new component *type* using the *runtime_support* with the given *targetgid*. This is a non-blocking call. The caller needs to call *future::get* on the result of this function to obtain the global id of the newly created object.

```
template<typename Component, typename ...Ts>
static naming::id_type create_component_colocated (naming::id_type const
&gid, Ts&&... vs)
```

Create a new component *type* using the *runtime_support* with the given *targetgid*. Block for the creation to finish.

```
template<typename Component>
static hpx::future<naming::id_type> copy_create_component_async (naming::id_type
const
&gid,
std::shared_ptr<Component>
const &p,
bool local_op)
```

```
template<typename Component>
static naming::id_type copy_create_component (naming::id_type const &gid,
std::shared_ptr<Component>
const &p, bool local_op)
```

```
template<typename Component>
```

```
static hpx::future<naming::id_type> migrate_component_async(naming::id_type
    const &target,
    get_locality,
    std::shared_ptr<Component>
    const &p, naming::id_type
    const
    &to_migrate)

template<typename Component, typename DistPolicy>
static hpx::future<naming::id_type> migrate_component_async(DistPolicy
    const &policy,
    std::shared_ptr<Component>
    const &p, naming::id_type
    const
    &to_migrate)

template<typename Component, typename Target>
static naming::id_type migrate_component(Target const &target, naming::id_type
    const &to_migrate,
    std::shared_ptr<Component> const
    &p)

static hpx::future<int> load_components_async(naming::id_type const &gid)

static int load_components(naming::id_type const &gid)

static hpx::future<void> call_startup_functions_async(naming::id_type
    const &gid, bool
    pre_startup)

static void call_startup_functions(naming::id_type const &gid, bool
    pre_startup)

static hpx::future<void> shutdown_async(naming::id_type const &targetgid, double
    timeout = -1)
    Shutdown the given runtime system.

static void shutdown(naming::id_type const &targetgid, double timeout = -1)

static void shutdown_all(naming::id_type const &targetgid, double timeout = -1)
    Shutdown the runtime systems of all localities.

static void shutdown_all(double timeout = -1)

static hpx::future<void> terminate_async(naming::id_type const &targetgid)
    Retrieve configuration information.

    Terminate the given runtime system

static void terminate(naming::id_type const &targetgid)

static void terminate_all(naming::id_type const &targetgid)
    Terminate the runtime systems of all localities.

static void terminate_all()

static void garbage_collect_non_blocking(naming::id_type const &target-
    gid)
```

```

static hpx::future<void> garbage_collect_async (naming::id_type const &targetgid)

static void garbage_collect (naming::id_type const &targetgid)

static hpx::future<naming::id_type> create_performance_counter_async (naming::id_type
tar-
get-
gid,
per-
for-
mance_counters::counter_
const
&info)

static naming::id_type create_performance_counter (naming::id_type
targetgid,          performance_counters::counter_info
const &info, error_code
&ec = throws)

static hpx::future<util::section> get_config_async (naming::id_type const &targetgid)
                                         Retrieve configuration information.

static void get_config (naming::id_type const &targetgid, util::section &ini)

static void remove_from_connection_cache_async (naming::id_type const
&target, naming::gid_type
const &gid,
parcelset::endpoints_type
const &endpoints)

```

segmented_algorithms

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

namespace segmented

Functions

```

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Optag_invoke (hpx::adjacent_difference_t,
ExPolicy &&pol-
icy,   FwdIter1
first,   FwdIter1
last,   FwdIter2
dest, Op &&op)

template<typename InIter1, typename InIter2, typename Optag_invoke (hpx::adjacent_difference_t, InIter1 first, InIter1 last, InIter2 dest, Op &&op)

```

namespace hpx

```
namespace segmented
```

Functions

```
template<typename InIter, typename Pred>
InIter tag_invoke(hpx::adjacent_find_t, InIter first, InIter last, Pred &&pred = Pred())

template<typename ExPolicy, typename SegIter, typename Pred>
hpx::parallel::util::detail::algorithm_result<ExPolicy, SegIter>::type tag_invoke(hpx::adjacent_find_t,
    ExPolicy
    &&policy,
    SegIter first,
    SegIter last,
    Pred &&pred)
```

```
namespace hpx
```

```
namespace segmented
```

Functions

```
template<typename InIter, typename F>
bool tag_invoke(hpx::none_of_t, InIter first, InIter last, F &&f)

template<typename ExPolicy, typename SegIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type tag_invoke(hpx::none_of_t,
    ExPolicy &&policy,
    SegIter first,
    SegIter last, F
    &&f)

template<typename InIter, typename F>
bool tag_invoke(hpx::any_of_t, InIter first, InIter last, F &&f)

template<typename ExPolicy, typename SegIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type tag_invoke(hpx::any_of_t,
    ExPolicy &&policy,
    SegIter first,
    SegIter last, F
    &&f)

template<typename InIter, typename F>
bool tag_invoke(hpx::all_of_t, InIter first, InIter last, F &&f)

template<typename ExPolicy, typename SegIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type tag_invoke(hpx::all_of_t,
    ExPolicy &&policy,
    SegIter first,
    SegIter last, F
    &&f)
```

```
namespace hpx
```

```
namespace segmented
```

Functions

```
template<typename InIter, typename T>
std::iterator_traits<InIter>::difference_type tag_invoke(hpx::count_t, InIter first, InIter last, T
                                                       const &value)

template<typename ExPolicy, typename SegIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<SegIter>::difference_type>::type tag

template<typename InIter, typename F>
std::iterator_traits<InIter>::difference_type tag_invoke(hpx::count_if_t, InIter first, InIter last, F
                                                       &&f)

template<typename ExPolicy, typename SegIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<SegIter>::difference_type>::type tag

namespace hpx

namespace segmented
```

Functions

```
template<typename InIter, typename OutIter, typename T, typename Op = std::plus<T>>
OutIter tag_invoke(hpx::exclusive_scan_t, InIter first, InIter last, OutIter dest, T init, Op &&op
= Op())

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Op = std::plus<T>>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type tag_invoke(hpx::exclusive_scan_t,
ExPolicy &&policy,
FwdIter1 first,
FwdIter1 last,
FwdIter2 dest, T
init, Op &&op =
Op())

namespace hpx
```

```
namespace segmented
```

Functions

```
template<typename SegIter, typename T>
SegIter tag_invoke(hpx::find_t, SegIter first, SegIter last, T const &val)

template<typename ExPolicy, typename SegIter, typename T>
parallel::util::detail::algorithm_result<ExPolicy, SegIter>::type tag_invoke(hpx::find_t, ExPolicy &&policy,
SegIter first, SegIter last, T const &val)

template<typename FwdIter, typename F>
FwdIter tag_invoke(hpx::find_if_t, FwdIter first, FwdIter last, F &&f)

template<typename ExPolicy, typename FwdIter, typename F>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type tag_invoke(hpx::find_if_t,
ExPolicy &&policy,
FwdIter first,
FwdIter last, F
&&f)

template<typename FwdIter, typename F>
FwdIter tag_invoke(hpx::find_if_not_t, FwdIter first, FwdIter last, F &&f)

template<typename ExPolicy, typename FwdIter, typename F>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type tag_invoke(hpx::find_if_not_t,
ExPolicy &&policy,
FwdIter first,
FwdIter last, F
&&f)
```

```
namespace hpx
```

```
namespace segmented
```

Functions

```
template<typename InIter, typename F>
InIter tag_invoke(hpx::for_each_t, InIter first, InIter last, F &&f)

template<typename ExPolicy, typename SegIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, SegIter>::type tag_invoke(hpx::for_each_t,
    ExPolicy
    &&policy,
    SegIter first,
    SegIter last, F
    &&f)

template<typename InIter, typename Size, typename F>
InIter tag_invoke(hpx::for_each_n_t, InIter first, Size count, F &&f)

template<typename ExPolicy, typename SegIter, typename Size, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, SegIter>::type tag_invoke(hpx::for_each_n_t,
    ExPolicy
    &&policy,
    SegIter first,
    Size count, F
    &&f)

namespace hpx
```

```
namespace segmented
```

Functions

```
template<typename SegIter, typename F>
SegIter tag_invoke(hpx::generate_t, SegIter first, SegIter last, F &&f)

template<typename ExPolicy, typename SegIter, typename F>
parallel::util::detail::algorithm_result<ExPolicy, SegIter>::type tag_invoke(hpx::generate_t,
    ExPolicy &&policy,
    SegIter first, SegIter
    last, F &&f)

namespace hpx
```

```
namespace segmented
```

Functions

```
template<typename InIter, typename OutIter, typename Op = std::plus<typename std::iterator_traits<InIter>::value_type>
OutIter tag_invoke(hpx::inclusive_scan_t, InIter first, InIter last, OutIter dest, Op &&op = Op())

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op = std::plus<typename std::iterator_traits<FwdIter1>::value_type>
FwdIter2 tag_invoke(hpx::exclusive_scan_t, FwdIter1 first, FwdIter1 last, FwdIter2 dest, Op &&op = Op())
```

```
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type tag_invoke(hpx::inclusive_scan_t,
    ExPolicy &&policy, FwdIter1 first,
    FwdIter1 last,
    FwdIter2 dest, Op
    &&op = Op())

template<typename InIter, typename OutIter, typename Op, typename T>
OutIter tag_invoke(hpx::inclusive_scan_t, InIter first, InIter last, OutIter dest, Op &&op, T
&&init)

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op, typename T>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type tag_invoke(hpx::inclusive_scan_t,
    ExPolicy &&policy, FwdIter1 first,
    FwdIter1 last,
    FwdIter2 dest, Op
    &&op, T &&init)

namespace hpx
```

```
namespace parallel
```

Typedefs

```
template<typename T>
using minmax_element_result = hpx::parallel::util::min_max_result<T>

namespace segmented
```

Typedefs

```
template<typename T>
using minmax_element_result = hpx::parallel::util::min_max_result<T>
```

Functions

```
template<typename SegIter, typename F>
SegIter tag_invoke(hpx::min_element_t, SegIter first, SegIter last, F &&f)

template<typename ExPolicy, typename SegIter, typename F>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, SegIter> tag_invoke(hpx::min_element_t,
    ExPolicy &&policy, SegIter first,
    SegIter last, F
    &&f)

template<typename SegIter, typename F>
SegIter tag_invoke(hpx::max_element_t, SegIter first, SegIter last, F &&f)

template<typename ExPolicy, typename SegIter, typename F>
```

```

    hpx::parallel::util::detail::algorithm_result_t<ExPolicy, SegIter> tag_invoke (hpx::max_element_t,
        ExPolicy &&policy, SegIter first,
        SegIter last, F
        &&f)

    template<typename SegIter, typename F>
    minmax_element_result<SegIter> tag_invoke (hpx::minmax_element_t, SegIter first, SegIter last,
        F &&f)

    template<typename ExPolicy, typename SegIter, typename F>
    hpx::parallel::util::detail::algorithm_result_t<ExPolicy, minmax_element_result<SegIter>> tag_invoke (hpx::minmax_e
        Ex-
        Pol-
        icy
        &&pol-
        icy,
        Se-
        gIter
        first,
        Se-
        gIter
        last,
        F
        &&f)

namespace hpx

namespace segmented

```

Functions

```

    template<typename InIterB, typename InIterE, typename T, typename F>
    T tag_invoke (hpx::reduce_t, InIterB first, InIterE last, T init, F &&f)

    template<typename ExPolicy, typename InIterB, typename InIterE, typename T, typename F>
    parallel::util::detail::algorithm_result<ExPolicy, T>::type tag_invoke (hpx::reduce_t, ExPolicy
        &&policy, InIterB first,
        InIterE last, T init, F
        &&f)

namespace hpx

namespace segmented

```

Functions

```
template<typename SegIter, typename OutIter, typename F>
hpx::parallel::util::in_out_result<SegIter, OutIter> tag_invoke(hpx::transform_t, SegIter first,
SegIter last, OutIter dest, F
&&f)
```

```
template<typename ExPolicy, typename SegIter, typename OutIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::parallel::util::in_out_result<SegIter, OutIter>>::type tag_invoke(hpx::transform_t, SegIter first, OutIter dest, F)
```



```
template<typename InIter1, typename InIter2, typename OutIter, typename F>
hpx::parallel::util::in_in_out_result<InIter1, InIter2, OutIter> tag_invoke(hpx::transform_t,
InIter1 first1, InIter1 last1, InIter2 first2,
OutIter dest, F &&f)
```

```
template<typename ExPolicy, typename InIter1, typename InIter2, typename OutIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::parallel::util::in_in_out_result<InIter1, InIter2, OutIter>>::type tag_invoke(hpx::transform_t, InIter1 first1, InIter1 last1, InIter2 first2,
OutIter dest, F)
```



```
template<typename InIter1, typename InIter2, typename OutIter, typename F>
```

```

hpx::parallel::util::in_in_out_result<InIter1, InIter2, OutIter> tag_invoke(hpx::transform_t,
InIter1 first1, InIter1
last1, InIter2 first2,
InIter2 last2, OutIter
dest, F &&f)
```

```

template<typename ExPolicy, typename InIter1, typename InIter2, typename OutIter, typename F>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::parallel::util::in_in_out_result<InIter1, InIter2, OutIter>>::type
```

```
namespace hpx
```

```
namespace segmented
```

Functions

```

template<typename InIter, typename OutIter, typename T, typename Op, typename Conv>
OutIter tag_invoke(hpx::transform_exclusive_scan_t, InIter first, InIter last, OutIter dest, T init,
Op &&op, Conv &&conv)
```

```

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Op, typename Conv>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type tag_invoke(hpx::transform_exclusive_scan_t,
ExPolicy &&pol-
icy, FwdIter1 first,
FwdIter1 last,
FwdIter2 dest, T
init, Op &&op,
Conv &&conv)
```

```
namespace hpx
```

```
namespace segmented
```

Functions

```
template<typename InIter, typename OutIter, typename Op, typename Conv>
OutIter tag_invoke(hpx::transform_inclusive_scan_t, InIter first, InIter last, OutIter dest, Op
&&op, Conv &&conv)

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op, typename Conv>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type tag_invoke(hpx::transform_inclusive_scan_t,
ExPolicy &&policy, FwdIter1 first,
FwdIter1 last,
FwdIter2 dest,
Op &&op, Conv
&&conv)

template<typename InIter, typename OutIter, typename T, typename Op, typename Conv>
OutIter tag_invoke(hpx::transform_inclusive_scan_t, InIter first, InIter last, OutIter dest, Op
&&op, Conv &&conv, T init)

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Op, typename Conv>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type tag_invoke(hpx::transform_inclusive_scan_t,
ExPolicy &&policy, FwdIter1 first,
FwdIter1 last,
FwdIter2 dest,
Op &&op, Conv
&&conv, T init)

namespace hpx

namespace segmented
```

Functions

```
template<typename SegIter, typename T, typename Reduce, typename Convert>
std::decay<T> tag_invoke(hpx::transform_reduce_t, SegIter first, SegIter last, T &&init, Reduce
&&red_op, Convert &&conv_op)

template<typename ExPolicy, typename SegIter, typename T, typename Reduce, typename Convert>
```

```

parallel::util::detail::algorithm_result<ExPolicy, typename std::decay<T>::type>::type tag_invoke(hpx::transform_re-
Ex-
Pol-
icy
&&pol-
icy,
Se-
gIter
first,
Se-
gIter
last,
T
&&init,
Re-
duce
&&red_op,
Con-
vert
&&conv_op)

template<typename FwdIter1, typename FwdIter2, typename T, typename Reduce, typename Convert>
T tag_invoke(hpx::transform_reduce_t, FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, T init,
Reduce &&red_op, Convert &&conv_op)

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Reduce, typename
parallel::util::detail::algorithm_result<ExPolicy, T>::type tag_invoke(hpx::transform_reduce_t,
ExPolicy &&policy,
FwdIter1 first1, FwdIter1
last1, FwdIter2 first2, T
init, Reduce &&red_op,
Convert &&conv_op)

namespace hpx

namespace util

namespace functional

struct segmented_iterator_begin

template<typename Iterator>
struct apply

```

Public Types

```
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_iterator
```

Public Functions

```
template<typename SegIter>
type operator()(SegIter iter) const

struct segmented_iterator_end
```

```
template<typename Iterator>
struct apply
```

Public Types

```
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_iterator
```

Public Functions

```
template<typename SegIter>
type operator()(SegIter iter) const

struct segmented_iterator_local
```

```
template<typename Iterator>
struct apply
```

Public Types

```
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_iterator
```

Public Functions

```
template<typename Iter>
type operator()(Iter iter) const

struct segmented_iterator_local_begin
```

```
template<typename Iterator>
struct apply
```

Public Types

```
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_raw_iterator
```

Public Functions

```
template<typename LocalSegIter>
type operator() (LocalSegIter iter) const
```

```
struct segmented_iterator_local_end
```

```
template<typename Iterator>
struct apply
```

Public Types

```
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_raw_iterator
```

Public Functions

```
template<typename LocalSegIter>
type operator() (LocalSegIter iter) const
```

```
struct segmented_iterator_segment
```

```
template<typename Iterator>
struct apply
```

Public Types

```
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::segment_iterator
```

Public Functions

```
template<typename Iter>
type operator() (Iter iter) const
```

```
template<typename ...Ts>
struct segmented_iterator_traits<util::zip_iterator<Ts...>, typename std::enable_if<util::all_of<typename segmenter>
```

Public Types

```
typedef std::true_type is_segmented_iterator
typedef util::zip_iterator<Ts...> iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::segment_iterator...> segment_iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_segment_iterator...> local_segment_iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_iterator...> local_iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_raw_iterator...> local_raw_iterator
```

Public Static Functions

```
static segment_iterator segment(iterator iter)
static local_iterator local(iterator iter)
static local_iterator begin(segment_iterator const &iter)
static local_iterator end(segment_iterator const &iter)
static local_raw_iterator begin(local_segment_iterator const &seg_iter)
static local_raw_iterator end(local_segment_iterator const &seg_iter)
static naming::id_type get_id(segment_iterator const &iter)
```

```
template<typename ...Ts>
```

```
struct segmented_local_iterator_traits<util::zip_iterator<Ts...>, typename std::enable_if<util::all_of<typename ...Ts>::value>::type> : public traits::iterator_traits<util::zip_iterator<Ts...>>
```

Public Types

```
typedef std::true_type is_segmented_local_iterator
typedef util::zip_iterator<typename segmented_local_iterator_traits<Ts>::iterator...> iterator
typedef util::zip_iterator<Ts...> local_iterator
typedef util::zip_iterator<typename segmented_local_iterator_traits<Ts>::local_raw_iterator...> local_raw_iterator
```

Public Static Functions

```
static local_raw_iterator local(local_iterator const &iter)
static local_iterator remote(local_raw_iterator const &iter)
```

```
namespace hpx
```

```
namespace traits
```

```
template<typename ...Ts>
struct segmented_iterator_traits<util::zip_iterator<Ts...>, typename std::enable_if<util::all_of<typename ...Ts>::value>::type> : public traits::iterator_traits<util::zip_iterator<Ts...>>
```

Public Types

```
typedef std::true_type is_segmented_iterator
typedef util::zip_iterator<Ts...> iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::segment_iterator...> segment_iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_segment_iterator...> local_segment_iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_iterator...> local_iterator
typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_raw_iterator...> local_raw_iterator
```

Public Static Functions

```
static segment_iterator segment (iterator iter)
static local_iterator local (iterator iter)
static local_iterator begin (segment_iterator const &iter)
static local_iterator end (segment_iterator const &iter)
static local_raw_iterator begin (local_segment_iterator const &seg_iter)
static local_raw_iterator end (local_segment_iterator const &seg_iter)
static naming::id_type get_id (segment_iterator const &iter)

template<typename ...Ts>
struct segmented_local_iterator_traits<util::zip_iterator<Ts...>, typename std::enable_if<util::all_of<Ts...>
```

Public Types

```
typedef std::true_type is_segmented_local_iterator
typedef util::zip_iterator<typename segmented_local_iterator_traits<Ts>::iterator...> iterator
typedef util::zip_iterator<Ts...> local_iterator
typedef util::zip_iterator<typename segmented_local_iterator_traits<Ts>::local_raw_iterator...> local_raw_iterator
```

Public Static Functions

```
static local_raw_iterator local (local_iterator const &iter)
static local_iterator remote (local_raw_iterator const &iter)

namespace functional

struct get_raw_iterator

template<typename Iterator>
struct apply
```

Public Functions

```
template<typename SegIter>
segmented_iterator_traits<Iterator>::local_raw_iterator operator() (SegIter iter) const
struct get_remote_iterator

template<typename Iterator>
struct apply
```

Public Functions

```
template<typename SegIter>
segmented_iterator_traits<Iterator>::local_iterator operator() (SegIter iter) const
```

statistics

See *Public API* for a list of names and headers that are part of the public HPX API.

```
namespace boost
```

```
namespace accumulators
```

```
namespace extract
```

Variables

```
const extractor<tag::histogram> histogram = {}
```

```
namespace tag
```

```
struct histogram : public depends_on<count>, public histogram_num_bins, public histogram_min_range
```

```
struct impl
```

```
template<typename Sample, typename Weight>
struct apply
```

Public Types

```
typedef hpx::util::detail::histogram_impl<Sample> type

namespace boost

namespace accumulators

namespace extract
```

Variables

```
const extractor<tag::rolling_max> rolling_max = {}

namespace tag

struct rolling_max : public depends_on<rolling_window>

struct impl

template<typename Sample, typename Weight>
struct apply
```

Public Types

```
typedef hpx::util::detail::rolling_max_impl<Sample> type

namespace boost

namespace accumulators

namespace extract

Variables

const extractor<tag::rolling_min> rolling_min = {}

namespace tag

struct rolling_min : public depends_on<rolling_window>

struct impl

template<typename Sample, typename Weight>
struct apply
```

Public Types

```
typedef hpx::util::detail::rolling_min_impl<Sample> type
```

2.9 Contributing to HPX

HPX development happens on Github. The following sections are a collection of useful information related to HPX development.

2.9.1 Contributing to HPX

The main source of information to understand the process of how to contribute to HPX can be found in [this document](#)²¹⁷. This is a living document that is constantly updated with relevant information.

2.9.2 HPX governance model

The HPX project is a meritocratic, consensus-based community project. Anyone with an interest in the project can join the community, contribute to the project design and participate in the decision making process. [This document](#)²¹⁸ describes how that participation takes place and how to set about earning merit within the project community.

2.9.3 Release procedure for HPX

Below is a step by step procedure for making an HPX release. We aim to produce two releases per year: one in March-April, and one in September-October.

This is a living document and may not be totally current or accurate. It is an attempt to capture current practices in making an HPX release. Please update it as appropriate.

One way to use this procedure is to print a copy and check off the lines as they are completed to avoid confusion.

1. Notify developers that a release is imminent.
2. For minor and major releases: create and check out a new branch at an appropriate point on `master` with the name `release-major.minor.X`. `major` and `minor` should be the major and minor versions of the release. For patch releases: check out the corresponding `release-major.minor.X` branch.
3. Write release notes in `docs/sphinx/releases/whats_new_$.VERSION.rst`. Keep adding merged PRs and closed issues to this until just before the release is made. Use `tools/generate_pr_issue_list.sh` to generate the lists. Add the new release notes to the table of contents in `docs/sphinx/releases.rst`.
4. Build the docs, and proof-read them. Update any documentation that may have changed, and correct any typos. Pay special attention to:
 - `$HPX_SOURCE/README.rst`
 - Update grant information
 - `docs/sphinx/releases/whats_new_$.VERSION.rst`
 - `docs/sphinx/about_hpx/people.rst`
 - Update collaborators

²¹⁷ <https://github.com/STELLAR-GROUP/hpx/blob/master/.github/CONTRIBUTING.md>

²¹⁸ <http://hpx.stellar-group.org/documents/governance/>

- Update grant information
5. This step does not apply to patch releases. For both APEX and libCDS:
 - Change the release branch to be the most current release tag available in the APEX/libCDS `git_external` section in the main `CMakeLists.txt`. Please contact the maintainers of the respective packages to generate a new release to synchronize with the HPX release (APEX²¹⁹, libCDS²²⁰).
 6. Make sure `HPX_VERSION_MAJOR/MINOR/SUBMINOR` in `CMakeLists.txt` contain the correct values. Change them if needed.
 7. This step does not apply to patch releases. Remove features which have been deprecated for at least 2 releases. This involves removing build options which enable those features from the main `CMakeLists.txt` and also deleting all related code and tests from the main source tree.
- The general deprecation policy involves a three-step process we have to go through in order to introduce a breaking change:
- a. First release cycle: add a build option that allows for explicitly disabling any old (now deprecated) code.
 - b. Second release cycle: turn this build option OFF by default.
 - c. Third release cycle: completely remove the old code.
- The main `CMakeLists.txt` contains a comment indicating for which version the breaking change was introduced first. In the case of deprecated features which don't have a replacement yet, we keep them around in case (like Vc for example).
8. Update the minimum required versions if necessary (compilers, dependencies, etc.) in `building_hpx.rst`.
 9. Verify that the Jenkins setups for the release branch on Rostam and Piz Daint are running and do not display any errors.
 10. Repeat the following steps until satisfied with the release.
 1. Change `HPX_VERSION_TAG` in `CMakeLists.txt` to `-rcN`, where N is the current iteration of this step. Start with `-rc1`.
 2. Create a pre-release on GitHub using the script `tools/roll_release.sh`. This script automatically tag with the corresponding release number. The script requires that you have the STELLAR Group signing key.
 3. This step is not necessary for patch releases. Notify `hpx-users@stellar-group.org` 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.
 11. 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.
 12. 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.

²¹⁹ <http://github.com/UO-OACISS/xpress-apex>

²²⁰ <https://github.com/STELLAR-GROUP/libcds>

13. Change HPX_VERSION_TAG in CMakeLists.txt to an empty string.
14. Add the release date to the caption of the current “What’s New” section in the docs, and change the value of HPX_VERSION_DATE in CMakeLists.txt.
15. Create a release on GitHub using the script tools/roll_release.sh. This script automatically tag the with the corresponding release number. The script requires that you have the STELLAR Group signing key.
16. Update the websites ([hpx.stellar-group.org²²¹](https://hpx.stellar-group.org), [stellar-group.org²²²](https://stellar-group.org) and [stellar.cct.lsu.edu²²³](https://stellar.cct.lsu.edu)). You can login on wordpress through *this page* <<https://hpx.stellar-group.org/wp-login.php>>. You can update the pages with the following:
 - Update links on the downloads page. Link to the release on GitHub.
 - Documentation links on the docs page (link to generated documentation on GitHub Pages). Follow the style of previous releases.
 - A new blog post announcing the release, which links to downloads and the “What’s New” section in the documentation (see previous releases for examples).
17. Merge release branch into master.
18. Post-release cleanup. Create a new pull request against master with the following changes:
 1. Modify the release procedure if necessary.
 2. Change HPX_VERSION_TAG in CMakeLists.txt back to -trunk.
 3. Increment HPX_VERSION_MINOR in CMakeLists.txt.
19. Update Vcpkg (<https://github.com/Microsoft/vcpkg>) to pull from latest release.
 - Update version number in CONTROL
 - Update tag and SHA512 to that of the new release
20. Update spack (<https://github.com/spack/spack>) with the latest HPX package.
 - Update version number in hpx/package.py and SHA256 to that of the new release
21. Announce the release on hpx-users@stellar-group.org, stellar@cct.lsu.edu, allcct@cct.lsu.edu, faculty@csc.lsu.edu, faculty@ece.lsu.edu, xpress@crest.iu.edu, the HPX Slack channel, the IRC channel, Sonia Sachs, our list of external collaborators, isocpp.org, reddit.com, HPC Wire, Inside HPC, Heise Online, and a CCT press release.
22. Beer and pizza.

2.9.4 Testing HPX

To ensure correctness of *HPX*, we ship a large variety of unit and regression tests. The tests are driven by the CTest²²⁴ tool and are executed automatically on each commit to the *HPX* Github²²⁵ 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* Issues²²⁶ tracker with detailed information about the target system.

²²¹ <https://hpx.stellar-group.org>

²²² <https://stellar-group.org>

²²³ <https://stellar.cct.lsu.edu>

²²⁴ <https://gitlab.kitware.com/cmake/community/wikis/doc/ctest/Testing-With-CTest>

²²⁵ <https://github.com/STELLAR-GROUP/hpx/>

²²⁶ <https://github.com/STELLAR-GROUP/hpx/issues>

Running tests manually

Running the tests manually is as easy as typing `make tests && make test`. This will build all tests and run them once the tests are built successfully. After the tests have been built, you can invoke separate tests with the help of the `ctest` command. You can list all available test targets using `make help | grep tests`. Please see the CTest Documentation²²⁷ for further details.

Running performance tests

We run performance tests on Piz Daint for each pull request using Jenkins. To run those performance tests locally or on Piz Daint, a script is provided under `tools/perftests_ci/local_run.sh` (to be run in the build directory specifying the *HPX* source directory as the argument to the script, default is `$HOME/projects/hpx_perftests_ci`.

Adding new performance tests

To add a new performance test, you need to wrap the portion of code to benchmark with `hpx::util::perftests_report`, passing the test name, the executor name and the function to time (can be a lambda). This facility is used to output the time results in a json format (format needed to compare the results and plot them). To effectively print them at the end of your test, call `hpx::util::perftests_print_times`. To see an example of use, see `future_overhead_report.cpp`. Finally, you can add the test to the CI report editing the `hpx_targets` variable for the executable name and the `hpx_test_options` variable for the corresponding options to use for the run in the performance test script `.jenkins/cscs-perftests/launch_perftests.sh`. And then run the `tools/perftests_ci/local_run.sh` script to get a reference json run (use the name of the test) to be added in the `tools/perftests_ci/perftest/references/daint_default` directory.

Issue tracker

If you stumble over a bug or missing feature in *HPX*, please submit an issue to our [HPX Issues²²⁸](#) page. For more information on how to submit support requests or other means of getting in contact with the developers, please see the [Support Website²²⁹](#) page.

Continuous testing

In addition to manual testing, we run automated tests on various platforms. We also run tests on all pull requests using both [CircleCI²³⁰](#) and a combination of [CDash²³¹](#) and [pycicle²³²](#). You can see the dashboards here: [CircleCI HPX dashboard²³³](#) and [CDash HPX dashboard²³⁴](#).

²²⁷ <https://www.cmake.org/cmake/help/latest/manual/ctest.1.html>

²²⁸ <https://github.com/STELLAR-GROUP/hpx/issues>

²²⁹ <https://stellar.cct.lsu.edu/support/>

²³⁰ <https://circleci.com>

²³¹ <https://www.kitware.com/cdash/project/about.html>

²³² <https://github.com/biddisco/pycicle/>

²³³ <https://circleci.com/gh/STELLAR-GROUP/hpx>

²³⁴ <https://cdash.cscs.ch/index.php?project=HPX>

2.9.5 Using docker for development

Although it can often be useful to set up a local development environment with system-provided or self-built dependencies, Docker²³⁵ provides a convenient alternative to quickly get all the dependencies needed to start development of *HPX*. Our testing setup on CircleCI²³⁶ uses a docker image to run all tests.

To get started you need to install Docker²³⁷ using whatever means is most convenient on your system. Once you have Docker²³⁸ installed, you can pull or directly run the docker image. The image is based on Debian and Clang, and can be found on Docker Hub²³⁹. To start a container using the *HPX* build environment, run:

```
$ docker run --interactive --tty stellargroup/build_env:latest bash
```

You are now in an environment where all the *HPX* build and runtime dependencies are present. You can install additional packages according to your own needs. Please see the Docker Documentation²⁴⁰ for more information on using Docker²⁴¹.

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)²⁴²).

2.9.6 Documentation

This documentation is built using Sphinx²⁴³, and an automatically generated API reference using Doxygen²⁴⁴ and Breathe²⁴⁵.

We always welcome suggestions on how to improve our documentation, as well as pull requests with corrections and additions.

Prerequisites

To build the *HPX* documentation, you need recent versions of the following packages:

- python3
- sphinx 3.5.4 (Python package)
- sphinx_rtd_theme (Python package)
- breathe 4.16.0 (Python package)
- doxygen

If the Python²⁴⁶ dependencies are not available through your system package manager, you can install them using the Python package manager pip:

²³⁵ <https://www.docker.com>

²³⁶ <https://circleci.com>

²³⁷ <https://www.docker.com>

²³⁸ <https://www.docker.com>

²³⁹ https://hub.docker.com/r/stellargroup/build_env/

²⁴⁰ <https://docs.docker.com/>

²⁴¹ <https://www.docker.com>

²⁴² <https://docs.docker.com/storage/bind-mounts/>

²⁴³ <http://www.sphinx-doc.org>

²⁴⁴ <https://www.doxygen.org>

²⁴⁵ <https://breathe.readthedocs.io/en/latest>

²⁴⁶ <https://www.python.org>

```
pip install --user sphinx sphinx_rtd_theme breathe
```

You may need to set the following CMake variables to make sure CMake can find the required dependencies.

DOXYGEN_ROOT:PATH

Specifies where to look for the installation of the Doxygen²⁴⁷ tool.

SPHINX_ROOT:PATH

Specifies where to look for the installation of the Sphinx²⁴⁸ tool.

BREATHE_APIDOC_ROOT:PATH

Specifies where to look for the installation of the Breathe²⁴⁹ tool.

Building documentation

Enable building of the documentation by setting `HPX_WITH_DOCUMENTATION=ON` during CMake²⁵⁰ configuration. To build the documentation, build the `docs` target using your build tool. The default output format is HTML documentation. You can choose alternative output formats (single-page HTML, PDF, and man) with the `HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS` CMake option.

Note: If you add new source files to the Sphinx documentation, you have to run CMake again to have the files included in the build.

Style guide

The documentation is written using reStructuredText. These are the conventions used for formatting the documentation:

- Use, at most, 80 characters per line.
- Top-level headings use over- and underlines with =.
- Sub-headings use only underlines with characters in decreasing level of importance: =, – and ..
- Use sentence case in headings.
- Refer to common terminology using :term:`Component`.
- Indent content of directives (.. directive::) by three spaces.
- For C++ code samples at the end of paragraphs, use :: and indent the code sample by 4 spaces.
 - For other languages (or if you don't want a colon at the end of the paragraph), use .. code-block:: language and indent by three spaces as with other directives.
- Use .. list-table:: to wrap tables with a lot of text in cells.

²⁴⁷ <https://www.doxygen.org>

²⁴⁸ <http://www.sphinx-doc.org>

²⁴⁹ <https://breathe.readthedocs.io/en/latest>

²⁵⁰ <https://www.cmake.org>

API documentation

The source code is documented using Doxygen. If you add new API documentation either to existing or new source files, make sure that you add the documented source files to the `doxygen_dependencies` variable in `docs/CMakeLists.txt`.

2.9.7 Module structure

This section explains the structure of an *HPX* module.

The tool `create_library_skeleton.py`²⁵¹ can be used to generate a basic skeleton. To create a library skeleton, run the tool in the `libs` subdirectory with the module name as an argument:

```
$ ./create_library_skeleton <lib_name>
```

This creates a skeleton with the necessary files for an *HPX* module. It will not create any actual source files. The structure of this skeleton is as follows:

- `<lib_name>/`
 - `README.rst`
 - `CMakeLists.txt`
 - `cmake`
 - `docs/`
 - * `index.rst`
 - `examples/`
 - * `CMakeLists.txt`
 - `include/`
 - * `hpx/`
 - `<lib_name>`
 - `src/`
 - * `CMakeLists.txt`
 - `tests/`
 - * `CMakeLists.txt`
 - * `unit/`
 - `CMakeLists.txt`
 - * `regressions/`
 - `CMakeLists.txt`
 - * `performance/`
 - `CMakeLists.txt`

²⁵¹ https://github.com/STELLAR-GROUP/hpx/blob/master/libs/create_library_skeleton.py

A README.rst should be always included which explains the basic purpose of the library and a link to the generated documentation.

A main CMakeLists.txt is created in the root directory of the module. By default it contains a call to add_hpx_module which takes care of most of the boilerplate required for a module. You only need to fill in the source and header files in most cases.

add_hpx_module requires a module name. Optional flags are:

Optional single-value arguments are:

- INSTALL_BINARIES: Install the resulting library.

Optional multi-value arguments-are:

- SOURCES: List of source files.
- HEADERS: List of header files.
- COMPAT_HEADERS: List of compatibility header files.
- DEPENDENCIES: Libraries that this module depends on, such as other modules.
- CMAKE_SUBDIRS: List of subdirectories to add to the module.

The include directory should contain only headers that other libraries need. For each of those headers, an automatic header test to check for self containment will be generated. Private headers should be placed under the src directory. This allows for clear separation. The cmake subdirectory may include additional CMake²⁵² scripts needed to generate the respective build configurations.

Compatibility headers (forwarding headers for headers whose location is changed when creating a module, if moving them from the main library) should be placed in an include_compatibility directory. This directory is not created by default.

Documentation is placed in the docs folder. A empty skeleton for the index is created, which is picked up by the main build system and will be part of the generated documentation. Each header inside the include directory will automatically be processed by Doxygen and included into the documentation. If a header should be excluded from the API reference, a comment // sphinx:undocumented needs to be added.

Tests are placed in suitable subdirectories of tests.

When in doubt, consult existing modules for examples on how to structure the module.

Finding circular dependencies

Our CI will perform a check to see if there are circular dependencies between modules. In cases where it's not clear what is causing the circular dependency, running the cpp-dependencies²⁵³ tool manually can be helpful. It can give you detailed information on exactly which files are causing the circular dependency. If you do not have the cpp-dependencies tool already installed, one way of obtaining it is by using our docker image. This way you will have exactly the same environment as on the CI. See *Using docker for development* for details on how to use the docker image.

To produce the graph produced by CI run the following command (HPX_SOURCE is assumed to hold the path to the HPX source directory):

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --graph-cycles circular_dependencies.dot
```

This will produce a dot file in the current directory. You can inspect this manually with a text editor. You can also convert this to an image if you have graphviz installed:

²⁵² <https://www.cmake.org>

²⁵³ <https://github.com/tomtom-international/cpp-dependencies>

```
$ dot circular_dependencies.dot -Tsvg -o circular_dependencies.svg
```

This produces an `svg` file in the current directory which shows the circular dependencies. Note that if there are no cycles the image will be empty.

You can use `cpp-dependencies` to print the include paths between two modules.

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --shortest <from> <to>
```

prints all possible paths from the module `<from>` to the module `<to>`. For example, as most modules depend on `config`, the following should give you a long list of paths from `algorithms` to `config`:

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --shortest algorithms config
```

The following should report that it can't find a path between the two modules:

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --shortest config algorithms
```

2.10 Releases

2.10.1 HPX V1.7.1 (Aug 12, 2021)

This is a bugfix release with a few minor fixes.

General changes

- Added a CMake option to assume that all types are bitwise serializable by default: `HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE`. The default value `OFF` corresponds to the old behaviour.
- Added a version check for Asio. The minimum Asio version supported by *HPX* is 1.12.0.
- Fixed a bug affecting usage of actions, where the internals of *HPX* relied on function addresses being unique. This was fixed by relying on variable addresses being unique instead.
- Made `hpx::util::bind` more strict in checking the validity of placeholders.
- Small performance improvement to spinlocks.
- Adapted the following parallel algorithms to C++20: `inclusive_scan`, `exclusive_scan`, `transform_inclusive_scan`, `transform_exclusive_scan`.

Breaking changes

- The experimental `hpx::execution::simdpar` execution policy (introduced in 1.7.0) was renamed to `hpx::execution::par_simd` for consistency with the other parallel policies.

Closed issues

- Issue #5494²⁵⁴ - Rename *simdpar* execution policy to *par_simd*
- Issue #5488²⁵⁵ - *hpx::util::bind* doesn't bounds-check placeholders
- Issue #5486²⁵⁶ - Possible V1.7.1 release

Closed pull requests

- PR #5500²⁵⁷ - Minor bug fix in transform exclusive and inclusive scan tests
- PR #5499²⁵⁸ - Rename *simdpar* to *par_simd*
- PR #5489²⁵⁹ - Adding bound-checking for bind placeholders
- PR #5485²⁶⁰ - Add Asio version check
- PR #5482²⁶¹ - Change extra archive data to rely on uniqueness of a variable address, not a function address
- PR #5448²⁶² - More fixes to enable for all types to be assumed to be bitwise copyable
- PR #5445²⁶³ - Improve performance of Spinlocks
- PR #5444²⁶⁴ - Adapt *transform_inclusive_scan* to C++ 20
- PR #5440²⁶⁵ - Adapt *transform_exclusive_scan* to C++ 20
- PR #5439²⁶⁶ - Adapt *inclusive_scan* to C++ 20
- PR #5436²⁶⁷ - Adapt *exclusive_scan* to C++20

2.10.2 HPX V1.7.0 (Jul 14, 2021)

This release is again focused on C++20 conformance of algorithms. Additionally, many new experimental sender-based algorithms have been added based on the latest proposals.

²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5494>

²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5488>

²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5486>

²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5500>

²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5499>

²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5489>

²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5485>

²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5482>

²⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5448>

²⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5445>

²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5444>

²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5440>

²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5439>

²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5436>

General changes

- The following algorithms have been adapted to be C++20 conformant:
 - `remove`,
 - `remove_if`,
 - `remove_copy`,
 - `remove_copy_if`,
 - `replace`,
 - `replace_if`,
 - `reverse`, and
 - `lexicographical_compare`.
- When the compiler and standard library support the standard execution policies `std::execution::seq`, `std::execution::par`, and `std::execution::par_unseq` they can now be used in all *HPX* parallel algorithms with equivalent behaviour to the non-task policies `hpx::execution::seq`, `hpx::execution::par`, and `hpx::execution::par_unseq`.
- Vc support has been fixed, after being broken in 1.6.0. In addition, *HPX* now experimentally supports GCC’s SIMD implementation, when available. The implementation can be used through the `hpx::execution::simd` and `hpx::execution::simdpar` execution policies.
- The customization points `sync_execute`, `async_execute`, `then_execute`, `post`, `bulk_sync_execute`, `bulk_async_execute`, and `bulk_then_execute` are now implemented using `tag_dispatch` (previously `tag_invoke`). Executors can still be implemented by providing the aforementioned functions as member functions of an executor.
- New functionality, enhancements, and fixes based on P0443r14 (executors proposal) and P1897 (sender-based algorithms) have been added to the `hpx::execution::experimental` namespace. These can be accessed through the `hpx/execution.hpp` and `hpx/local/execution.hpp` headers. In particular, the following sender-based algorithms have been added:
 - `detach`,
 - `ensure_started`,
 - `just`,
 - `just_on`,
 - `let_error`,
 - `let_value`,
 - `on`,
 - `transform`, and
 - `when_all`.

Additionally, futures now implement the sender concept. `make_future` can be used to turn a sender into a future. All functionality is experimental and can change without notice.

- All `hpx::init` and `hpx::start` overloads now take `std::functions` instead of `hpx::util::function_nons`. No changes should be required in user code to accommodate this change.

- `hpx::util::unwrapping` and other related unwrapping functionality has been moved up into the `hpx` namespace. Names in `hpx::util` are still usable with a deprecation warning. This functionality can now be accessed through the `hpx/unwrap.hpp` and `hpx/local/unwrap.hpp` headers.
- The default tag for APEX has been update from 2.3.1 to 2.4.0. In particular, this fixes a bug which could lead to hangs in distributed runs.
- The dependency on Boost.Asio has been replaced with the standalone Asio available at <https://github.com/chriskohlhoff/asio>. By default, a system-installed Asio will be used. `ASIO_ROOT` can be given as a hint to tell CMake where to find Asio. Alternatively, Asio can be fetched automatically using CMake's `fetchcontent` by setting `HPX_WITH_FETCH_ASIO=ON`. In general, dependencies on Boost have again been reduced.
- Modularization of the library has continued. In this release almost all functionality has been moved into modules. These changes do not generally affect user code. Warnings are still issued for headers that have moved.
- `hipBLAS` is now optional when compiling with `hipcc`. A warning instead of an error will be printed if `hipBLAS` is not found during configuration.
- Previously `HPX_COMPUTE_HOST_CODE` was defined in host code only if HPX was configured with CUDA or HIP. In this release `HPX_COMPUTE_HOST_CODE` is always defined in host code.
- An experimental `HPX_WITH_PRECOMPILED_HEADERS` CMake option has been added to use precompiled headers when building HPX. This option should not be used on Windows.
- Numerous bug fixes.

Breaking changes

- The minimum required CMake version is now 3.17.
- The minimum required Boost version is now 1.71.0.
- The customization mechanism used to implement and extend sender functionality and algorithms has been renamed from `tag_invoke` to `tag_dispatch`. All customization of sender functionality should be done by overloading `tag_dispatch`.
- The following compatibility options have been removed, along with their compatibility implementations:
 - `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY`
 - `HPX_WITH_ACTION_BASE_COMPATIBILITY`
 - `HPX_WITH_EMBEDDED_THREAD_POOLS_COMPATIBILITY`
 - `HPX_WITH_POOL_EXECUTOR_COMPATIBILITY`
 - `HPX_WITH_PROMISE_ALIAS_COMPATIBILITY`
 - `HPX_WITH_REGISTER_THREAD_COMPATIBILITY`
 - `HPX_WITH_REGISTER_THREAD_OVERLOADS_COMPATIBILITY`
 - `HPX_WITH_THREAD_AWARE_TIMER_COMPATIBILITY`
 - `HPX_WITH_THREAD_EXECUTORS_COMPATIBILITY`
 - `HPX_WITH_THREAD_POOL_OS_EXECUTOR_COMPATIBILITY`
- The `HPX_WITH_THREAD_SCHEDULERS` CMake option has been removed. All schedulers are now enabled when possible.
- `HPX_WITH_INIT_START_OVERLOADS_COMPATIBILITY` has been turned off by default.

Closed issues

- Issue #5423²⁶⁸ - Fix lvalue-ref qualified connect for when_all-sender
- Issue #5412²⁶⁹ - Link error
- Issue #5397²⁷⁰ - Performance regression in thread annotations
- Issue #5395²⁷¹ - HPX 1.7.0-rc1 fails to build icw APEX + OTF2
- Issue #5385²⁷² - HPX 1.7 crashes on Piz Daint > 64 nodes
- Issue #5380²⁷³ - CMake should search for asio package installed on the system
- Issue #5378²⁷⁴ - HPX 1.7.0 stopped building on Fedora
- Issue #5369²⁷⁵ - HPX 1.6 and master hangs on Summit for > 64 nodes
- Issue #5358²⁷⁶ - HPX init fails for single-core environments
- Issue #5345²⁷⁷ - Rename P2220 property CPOs?
- Issue #5333²⁷⁸ - HPX does not compile on the new Mac OSX using the M1 chip
- Issue #5317²⁷⁹ - Consider making hipblas optional
- Issue #5306²⁸⁰ - asio fails to build with CUDA 10.0
- Issue #5294²⁸¹ - execution::on should be based on execution::schedule
- Issue #5275²⁸² - HPX V1.6.0 fails on Fedora release
- Issue #5270²⁸³ - HPX-1.6.0 fails to build on Windows 10
- Issue #5257²⁸⁴ - Allow triggering the output of OS thread affinity from configuration settings
- Issue #5246²⁸⁵ - HPX fails to build on ppc64le
- Issue #5232²⁸⁶ - Annotation using hpx::util::annotated_function not working
- Issue #5222²⁸⁷ - Build and link errors with ittnotify enabled
- Issue #5204²⁸⁸ - Move algorithms to tag_fallback_dispatch
- Issue #5163²⁸⁹ - Remove module-specific compatibility and deprecation options

²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5423>

²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5412>

²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5397>

²⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/5395>

²⁷² <https://github.com/STELLAR-GROUP/hpx/issues/5385>

²⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/5380>

²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5378>

²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5369>

²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5358>

²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5345>

²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5333>

²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5317>

²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5306>

²⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/5294>

²⁸² <https://github.com/STELLAR-GROUP/hpx/issues/5275>

²⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/5270>

²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5257>

²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5246>

²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5232>

²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5222>

²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5204>

²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5163>

- Issue #5161²⁹⁰ - Bump required CMake version to 3.17
- Issue #5143²⁹¹ - Searching for HPX-Application to generate work on multiple Nodes

Closed pull requests

- PR #5438²⁹² - Delete datapar/foreach_tests.hpp
- PR #5437²⁹³ - Add back explicit -pthread flags when available
- PR #5435²⁹⁴ - This adds support for systems that assume all types are bitwise serializable by default
- PR #5434²⁹⁵ - Update CUDA polling logging to be more verbose
- PR #5433²⁹⁶ - Fix when_all_sender connect for references
- PR #5432²⁹⁷ - Add deprecation warnings for v1.8
- PR #5431²⁹⁸ - Rename the new P0443/P2300 executor to thread_pool_scheduler
- PR #5430²⁹⁹ - Revert “Adding the missing defined for HPX_HAVE_DEPRECATED_WARNINGS”
- PR #5427³⁰⁰ - Removing unneeded typedef
- PR #5426³⁰¹ - Adding more concept checks for sender/receiver algorithms
- PR #5425³⁰² - Adding the missing defined for HPX_HAVE_DEPRECATED_WARNINGS
- PR #5424³⁰³ - Disable Vc in final docker image created in CI
- PR #5422³⁰⁴ - Adding execution::experimental::bulk algorithm
- PR #5420³⁰⁵ - Update logic to find threading library
- PR #5418³⁰⁶ - Reduce max size and number of files in ccache cache
- PR #5417³⁰⁷ - Final release notes for 1.7.0
- PR #5416³⁰⁸ - Adapt uninitialized_value_construct and uninitialized_value_construct_n to C++ 20
- PR #5415³⁰⁹ - Adapt uninitialized_default_construct and uninitialized_default_construct_n to C++ 20
- PR #5414³¹⁰ - Improve integration of futures and senders

²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5161>

²⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/5143>

²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5438>

²⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5437>

²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5435>

²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5434>

²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5433>

²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5432>

²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5431>

²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5430>

³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5427>

³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5426>

³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5425>

³⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5424>

³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5422>

³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5420>

³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5418>

³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5417>

³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5416>

³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5415>

³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5414>

- PR #5413³¹¹ - Fixing sender/receiver code base to compile with MSVC
- PR #5407³¹² - Handle exceptions thrown during initialization of parcel handler
- PR #5406³¹³ - Simplify dispatching to annotation handlers
- PR #5405³¹⁴ - Fetch Asio automatically in perftests CI
- PR #5403³¹⁵ - Create generic executor that adds annotations to any other executor
- PR #5402³¹⁶ - Adapt uninitialized_fill and uninitialized_fill_n to C++ 20
- PR #5401³¹⁷ - Modernize a variety of facilities related to parallel algorithms
- PR #5400³¹⁸ - Fix sliding semaphore test
- PR #5399³¹⁹ - Rename leftover tagFallback_invoke to tagFallback_dispatch
- PR #5398³²⁰ - Improve logging in AGAS symbol namespace
- PR #5396³²¹ - Introduce compatibility layer for collective operations
- PR #5394³²² - Enable OTF2 in APEX CI configuration
- PR #5393³²³ - Update APEX tag
- PR #5392³²⁴ - Fixing wrong usage of std::forward
- PR #5391³²⁵ - Fix forwarding in transform_receiver constructor
- PR #5390³²⁶ - Make sure shared priority scheduler steals tasks on the current NUMA domain when (core) stealing is enabled
- PR #5389³²⁷ - Adapt uninitialized_move and uninitialized_move_n to C++ 20
- PR #5388³²⁸ - Fixing gather_there for used with lvalue reference argument
- PR #5387³²⁹ - Extend thread state logging and change default stealing parameters
- PR #5386³³⁰ - Attempt to fix the startup hang with nodes > 32
- PR #5384³³¹ - Remove HPX 1.5.0 deprecations
- PR #5382³³² - Prefer installed Asio before considering FetchContent
- PR #5379³³³ - Allow using pre-downloaded (not installed) versions of Asio and/or Apex

³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5413>

³¹² <https://github.com/STELLAR-GROUP/hpx/pull/5407>

³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5406>

³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5405>

³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5403>

³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5402>

³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5401>

³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5400>

³¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5399>

³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5398>

³²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5396>

³²² <https://github.com/STELLAR-GROUP/hpx/pull/5394>

³²³ <https://github.com/STELLAR-GROUP/hpx/pull/5393>

³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5392>

³²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5391>

³²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5390>

³²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5389>

³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5388>

³²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5387>

³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5386>

³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5384>

³³² <https://github.com/STELLAR-GROUP/hpx/pull/5382>

³³³ <https://github.com/STELLAR-GROUP/hpx/pull/5379>

- PR #5376³³⁴ - Remove unnecessary explicit listing of library modules.rst files in CMakeLists.txt
- PR #5375³³⁵ - Slight performance improvement for `hpx::copy` and `hpx::move` et.al.
- PR #5374³³⁶ - Remove unnecessary moves from future sender implementations
- PR #5373³³⁷ - More changes to clang-cuda Jenkins configuration
- PR #5372³³⁸ - Slight improvements to `min/max/minmax_element` algorithms
- PR #5371³³⁹ - Adapt `uninitialized_copy` and `uninitialized_copy_n` to C++ 20
- PR #5370³⁴⁰ - Decay types in `just_sender value_types` to match stored types
- PR #5367³⁴¹ - Disable `pkgconfig` by default again on macOS
- PR #5365³⁴² - Use `ccache` for Jenkins builds on Piz Daint
- PR #5363³⁴³ - Update cudatoolkit module name in clang-cuda Jenkins configuration
- PR #5362³⁴⁴ - Adding `channel_communicator`
- PR #5361³⁴⁵ - Fix compilation with MPI enabled
- PR #5360³⁴⁶ - Update APEX and asio tags
- PR #5359³⁴⁷ - Fix check for pu-step in single-core case
- PR #5357³⁴⁸ - Making sure collective operations can be reused by preallocating communicator
- PR #5356³⁴⁹ - Update API documentation
- PR #5355³⁵⁰ - Make the `sequenced_executor processing_units_count` member function const
- PR #5354³⁵¹ - Making sure `default_stack_size` is defined whenever declared
- PR #5353³⁵² - Add CUDA timestamp support to HPX Hardware Clock
- PR #5352³⁵³ - Adding missing includes
- PR #5351³⁵⁴ - Adding `enable_logging/disable_logging` API functions
- PR #5350³⁵⁵ - Adapt `lexicographical_compare` to C++20
- PR #5349³⁵⁶ - Update minimum boost version needed on the docs

³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5376>³³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5375>³³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5374>³³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5373>³³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5372>³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5371>340 <https://github.com/STELLAR-GROUP/hpx/pull/5370>341 <https://github.com/STELLAR-GROUP/hpx/pull/5367>342 <https://github.com/STELLAR-GROUP/hpx/pull/5365>343 <https://github.com/STELLAR-GROUP/hpx/pull/5363>344 <https://github.com/STELLAR-GROUP/hpx/pull/5362>345 <https://github.com/STELLAR-GROUP/hpx/pull/5361>346 <https://github.com/STELLAR-GROUP/hpx/pull/5360>347 <https://github.com/STELLAR-GROUP/hpx/pull/5359>348 <https://github.com/STELLAR-GROUP/hpx/pull/5357>349 <https://github.com/STELLAR-GROUP/hpx/pull/5356>350 <https://github.com/STELLAR-GROUP/hpx/pull/5355>351 <https://github.com/STELLAR-GROUP/hpx/pull/5354>352 <https://github.com/STELLAR-GROUP/hpx/pull/5353>353 <https://github.com/STELLAR-GROUP/hpx/pull/5352>354 <https://github.com/STELLAR-GROUP/hpx/pull/5351>355 <https://github.com/STELLAR-GROUP/hpx/pull/5350>356 <https://github.com/STELLAR-GROUP/hpx/pull/5349>

- PR #5348³⁵⁷ - Rename `tag_invoke` and related facilities to `tag_dispatch`
- PR #5347³⁵⁸ - Remove `make_` prefix for executor properties
- PR #5346³⁵⁹ - Remove and disable compatibility options for 1.7.0
- PR #5343³⁶⁰ - Fix `timed_executor` static cast conversion
- PR #5342³⁶¹ - Refactor CUDA event polling
- PR #5341³⁶² - Adding `make_with_annotation` and `get_annotation` properties
- PR #5339³⁶³ - Making sure `hpx::util::hardware::timestamp()` is always defined
- PR #5338³⁶⁴ - Fixing `timed_executor` specializations of customization points
- PR #5335³⁶⁵ - Make `partial_algorithm` work with any number of arguments
- PR #5334³⁶⁶ - Follow up `iter_sent` include on #5225
- PR #5332³⁶⁷ - Simplify `tag_invoke` and friends
- PR #5331³⁶⁸ - More work on cleaning up executor CPOs
- PR #5330³⁶⁹ - Add option to disable `pkgconfig` generation
- PR #5328³⁷⁰ - Adapt data parallel support using `std::simd`
- PR #5327³⁷¹ - Fix missing `ifdef HPX_SMT_PAUSE`
- PR #5326³⁷² - Adding `resize()` to `serialize_buffer` allowing to shrink its size
- PR #5324³⁷³ - Add get member functions to `async_rw_mutex` proxy objects for explicitly getting the wrapped value
- PR #5323³⁷⁴ - Add `keep_future` algorithm
- PR #5322³⁷⁵ - Replace executor customization point implementations with `tag_invoke`
- PR #5321³⁷⁶ - Separate segmented algorithms for reduce
- PR #5320³⁷⁷ - Fix `is_sender` trait and other small fixes to p0443 traits
- PR #5319³⁷⁸ - gcc 11.1 c++20 build fixes
- PR #5318³⁷⁹ - Make `hipblas` dependency optional as not always available

³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5348>

³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5347>

³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5346>

³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5343>

³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5342>

³⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5341>

³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5339>

³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5338>

³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5335>

³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5334>

³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5332>

³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5331>

³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5330>

³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5328>

³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/5327>

³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/5326>

³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/5324>

³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5323>

³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5322>

³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5321>

³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5320>

³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5319>

³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5318>

- PR #5316³⁸⁰ - Attempt to fix checking for libatomic
- PR #5315³⁸¹ - Add explicit keyword to fixture constructor
- PR #5314³⁸² - Fix a race condition in async mpi affecting limiting executor
- PR #5312³⁸³ - Use local runtime and local headers in local-only modules and tests
- PR #5311³⁸⁴ - Add GCC 11 builder to jenkins
- PR #5310³⁸⁵ - Adding `hpx::execution::experimental::task_group`
- PR #5309³⁸⁶ - Separate datapar
- PR #5308³⁸⁷ - Separate segmented algorithms for `find`, `find_if`, `find_if_not`
- PR #5307³⁸⁸ - Separate segmented algorithms for `fill` and `generate`
- PR #5304³⁸⁹ - Fix compilation of sender CPOs with nvcc
- PR #5300³⁹⁰ - Remove PRIVATE flag that was propagated into the LANGUAGES
- PR #5298³⁹¹ - Separate datapar
- PR #5297³⁹² - Specify exact cmake and ninja versions when loading them in jenkins jobs
- PR #5295³⁹³ - Update clang-newest configuration to use clang 12 and Boost 1.76.0
- PR #5293³⁹⁴ - Fix Clang 11 cuda_future test bug
- PR #5292³⁹⁵ - Add `async_rw_mutex` based on senders
- PR #5291³⁹⁶ - “Fix” termination detection
- PR #5290³⁹⁷ - Fixed source file line statements in examples documentation
- PR #5289³⁹⁸ - Allow splitting of futures holding `std::tuple`
- PR #5288³⁹⁹ - Move algorithms to `tag_fallback_invoke`
- PR #5287⁴⁰⁰ - Move algorithms to `tag_fallback_invoke`
- PR #5285⁴⁰¹ - Fix clang-format failure on master
- PR #5284⁴⁰² - Replacing `util::function_nonser` on `std::function` in `hpx_init`

³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5316>³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5315>³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5314>³⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5312>³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5311>³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5310>386 <https://github.com/STELLAR-GROUP/hpx/pull/5309>387 <https://github.com/STELLAR-GROUP/hpx/pull/5308>388 <https://github.com/STELLAR-GROUP/hpx/pull/5307>389 <https://github.com/STELLAR-GROUP/hpx/pull/5304>390 <https://github.com/STELLAR-GROUP/hpx/pull/5300>391 <https://github.com/STELLAR-GROUP/hpx/pull/5298>392 <https://github.com/STELLAR-GROUP/hpx/pull/5297>393 <https://github.com/STELLAR-GROUP/hpx/pull/5295>394 <https://github.com/STELLAR-GROUP/hpx/pull/5293>395 <https://github.com/STELLAR-GROUP/hpx/pull/5292>396 <https://github.com/STELLAR-GROUP/hpx/pull/5291>397 <https://github.com/STELLAR-GROUP/hpx/pull/5290>398 <https://github.com/STELLAR-GROUP/hpx/pull/5289>399 <https://github.com/STELLAR-GROUP/hpx/pull/5288>400 <https://github.com/STELLAR-GROUP/hpx/pull/5287>401 <https://github.com/STELLAR-GROUP/hpx/pull/5285>402 <https://github.com/STELLAR-GROUP/hpx/pull/5284>

- PR #5282⁴⁰³ - Update Boost for daint 20.11 after update
- PR #5281⁴⁰⁴ - Fix Segmentation fault on `foreach_datapar_zipiter`
- PR #5280⁴⁰⁵ - Avoid modulo by zero in `counting_iterator` test
- PR #5279⁴⁰⁶ - Fix more GCC 10 deprecation warnings
- PR #5277⁴⁰⁷ - Small fixes and improvements to CUDA/MPI polling
- PR #5276⁴⁰⁸ - Fix typo in docs
- PR #5274⁴⁰⁹ - More P1897 algorithms
- PR #5273⁴¹⁰ - Retry CDash submissions on failure
- PR #5272⁴¹¹ - Fix bogus deprecation warnings with GCC 10
- PR #5271⁴¹² - Correcting target ids for `symbol_namespace::iterate`
- PR #5268⁴¹³ - Adding generic `require`, `require_concept`, and `query` properties
- PR #5267⁴¹⁴ - Support annotations in `hpx::transform_reduce`
- PR #5266⁴¹⁵ - Making late command line options available for local runtime
- PR #5265⁴¹⁶ - Leverage `no_unique_address` for `member_pack`
- PR #5264⁴¹⁷ - Adopt format in more places
- PR #5262⁴¹⁸ - Install HPX in Rostam Jenkins jobs
- PR #5261⁴¹⁹ - Limit Rostam Jenkins jobs to marvin partition temporarily
- PR #5260⁴²⁰ - Separate segmented algorithms for `transform_reduce`
- PR #5259⁴²¹ - Making sure late command line options are recognized as configuration options
- PR #5258⁴²² - Allow for HPX algorithms being invoked with std execution policies
- PR #5256⁴²³ - Separate segmented algorithms for `transform`
- PR #5255⁴²⁴ - Future/sender adapters
- PR #5254⁴²⁵ - Fixing datapar

⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5282>

⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5281>

⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5280>

⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5279>

⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5277>

⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5276>

⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5274>

⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5273>

⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5272>

⁴¹² <https://github.com/STELLAR-GROUP/hpx/pull/5271>

⁴¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5268>

⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5267>

⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5266>

⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5265>

⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5264>

⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5262>

⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5261>

⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5260>

⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5259>

⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/5258>

⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/5256>

⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5255>

⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5254>

- PR #5253⁴²⁶ - Add utility to format ranges
- PR #5252⁴²⁷ - Remove uses of Boost.Bimap
- PR #5251⁴²⁸ - Banish <iostream> from library headers
- PR #5250⁴²⁹ - Try fixing vc circle ci
- PR #5249⁴³⁰ - Adding missing header
- PR #5248⁴³¹ - Use old Piz Daint modules after upgrade
- PR #5247⁴³² - Significantly speedup simple for_each, for_loop, and transform
- PR #5245⁴³³ - P1897 operator| overloads
- PR #5244⁴³⁴ - P1897 when_all
- PR #5243⁴³⁵ - Make sure HPX_DEBUG is set based on HPX's build type, not consuming project's build type
- PR #5242⁴³⁶ - Moving last files unrelated to parcel layer to modules
- PR #5240⁴³⁷ - change namespace for transform_loop.hpp
- PR #5238⁴³⁸ - Make sure annotations are used in the binary transform
- PR #5237⁴³⁹ - Add P1897 just, just_on, and on algorithms
- PR #5236⁴⁴⁰ - Add an example demonstrating the use of the invoke_function_action facility
- PR #5235⁴⁴¹ - Attempting to fix datapar compilation issues
- PR #5234⁴⁴² - Fix small typo in --hpx:local option description
- PR #5233⁴⁴³ - Only find Boost.Iostreams if required for plugins
- PR #5231⁴⁴⁴ - Sort printed config options
- PR #5230⁴⁴⁵ - Fix C++20 replace algo adaptation misses
- PR #5229⁴⁴⁶ - Remove leftover Boost include from sync_wait.hpp
- PR #5228⁴⁴⁷ - Print module name only if it has custom configuration settings
- PR #5227⁴⁴⁸ - Update .codespell_whitelist

⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5253>

⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5252>

⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5251>

⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5250>

⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5249>

⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5248>

⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/5247>

⁴³³ <https://github.com/STELLAR-GROUP/hpx/pull/5245>

⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5244>

⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5243>

⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5242>

⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5240>

⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5238>

⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5237>

⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5236>

⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5235>

⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5234>

⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5233>

⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5231>

⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5230>

⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5229>

⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5228>

⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5227>

- PR #5226⁴⁴⁹ - Use new docker image in all CircleCI steps
- PR #5225⁴⁵⁰ - Adapt reverse to C++20
- PR #5224⁴⁵¹ - Separate segmented algorithms for `none_of`, `any_of` and `all_of`
- PR #5223⁴⁵² - Fixing build system for `itnotify`
- PR #5221⁴⁵³ - Moving LCO related files to modules
- PR #5220⁴⁵⁴ - Separate segmented algorithms for `count` and `count_if`
- PR #5218⁴⁵⁵ - Separate segmented algorithms for `adjacent_find`
- PR #5217⁴⁵⁶ - Add a HIP github action
- PR #5215⁴⁵⁷ - Update ROCm to 4.0.1 on Rostam
- PR #5214⁴⁵⁸ - Fix clang-format error in `sender.hpp`
- PR #5213⁴⁵⁹ - Removing ESSENTIAL option to the doc example
- PR #5212⁴⁶⁰ - Separate segmented algorithms for `for_each_n`
- PR #5211⁴⁶¹ - Minor adapted algos fixes
- PR #5210⁴⁶² - Fixing `is_invocable` deprecation warnings
- PR #5209⁴⁶³ - Moving more files into modules (actions, components, `init_runtime`, etc.)
- PR #5208⁴⁶⁴ - Add examples and explanation on when `tag_fallback/priority` are useful
- PR #5207⁴⁶⁵ - Always define `HPX_COMPUTE_HOST_CODE` for host code
- PR #5206⁴⁶⁶ - Add formatting exceptions for `libhpx` to `create_module_skeleton.py`
- PR #5205⁴⁶⁷ - Moving all distribution policies into modules
- PR #5203⁴⁶⁸ - Move copy algorithms to `tag_fallback_invoke`
- PR #5202⁴⁶⁹ - Make `HPX_WITH_PSEUDO_DEPENDENCIES` a cache variable
- PR #5201⁴⁷⁰ - Replaced `tag_invoke` with `tag_fallback_invoke` for `adjacent_find` algorithm
- PR #5200⁴⁷¹ - Moving files to (distributed) runtime module

⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5226>

⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5225>

⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/5224>

⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/5223>

⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/5221>

⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5220>

⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5218>

⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5217>

⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5215>

⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5214>

⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5213>

⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5212>

⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5211>

⁴⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5210>

⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5209>

⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5208>

⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5207>

⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5206>

⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5205>

⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5203>

⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5202>

⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5201>

⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/5200>

- PR #5199⁴⁷² - Update ICC module name on Piz Daint Jenkins configuration
- PR #5198⁴⁷³ - Add doxygen documentation for thread_schedule_hint
- PR #5197⁴⁷⁴ - Attempt to fix compilation of context implementations with unity build enabled
- PR #5196⁴⁷⁵ - Re-enable component tests
- PR #5195⁴⁷⁶ - Moving files related to colocation logic
- PR #5194⁴⁷⁷ - Another attempt at fixing the Fedora 35 problem
- PR #5193⁴⁷⁸ - Components module
- PR #5192⁴⁷⁹ - Adapt replace (_if) to C++20
- PR #5190⁴⁸⁰ - Set compatibility headers by default to on
- PR #5188⁴⁸¹ - Bump Boost minimum version to 1.71.0
- PR #5187⁴⁸² - Force CMake to set the -std=c++XX flag
- PR #5186⁴⁸³ - Remove message to print .cu extension whenever .cu files are encountered
- PR #5185⁴⁸⁴ - Remove some minor unnecessary CMake options
- PR #5184⁴⁸⁵ - Remove some leftover HPX_WITH_*_SCHEDULER uses
- PR #5183⁴⁸⁶ - Remove dependency on boost/iterators/iterator_categories.hpp
- PR #5182⁴⁸⁷ - Fixing Fedora 35 for Power architectures
- PR #5181⁴⁸⁸ - Bump version number and tag post 1.6.0 release
- PR #5180⁴⁸⁹ - Fix htts_v2 tests linking
- PR #5179⁴⁹⁰ - Make sure --hpx:local command line option is respected with networking is off but distributed runtime is on
- PR #5177⁴⁹¹ - Remove module cmake options
- PR #5176⁴⁹² - Starting to separate segmented algorithms: for_each
- PR #5174⁴⁹³ - Don't run segmented algorithms twice on CircleCI
- PR #5173⁴⁹⁴ - Fetching APEX using cmake FetchContent

⁴⁷² <https://github.com/STELLAR-GROUP/hpx/pull/5199>⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/5198>⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5197>⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5196>⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5195>⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5194>⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5193>⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5192>⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5190>⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5188>⁴⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5187>⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5186>⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5185>⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5184>⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5183>⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5182>⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5181>⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5180>⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5179>⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5177>⁴⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5176>⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5174>⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5173>

- PR #5172⁴⁹⁵ - Add separate local-only entry point
- PR #5171⁴⁹⁶ - Remove HPX_WITH_THREAD_SCHEDULERS CMake option
- PR #5170⁴⁹⁷ - Add HPX_WITH_PRECOMPILED_HEADERS option
- PR #5166⁴⁹⁸ - Moving some action tests to modules
- PR #5165⁴⁹⁹ - Require cmake 3.17
- PR #5164⁵⁰⁰ - Move thread_pool_suspension_helper files to small utility module
- PR #5160⁵⁰¹ - Adding checks ensuring modules are not cross-referenced from other module categories
- PR #5158⁵⁰² - Replace boost::asio with standalone asio
- PR #5155⁵⁰³ - Allow logging when distributed runtime is off
- PR #5153⁵⁰⁴ - Components module
- PR #5152⁵⁰⁵ - Move more files to performance counter module
- PR #5150⁵⁰⁶ - Adapt remove_copy (_if) to C++20
- PR #5144⁵⁰⁷ - AGAS module
- PR #5125⁵⁰⁸ - Adapt remove and remove_if to C++20
- PR #5117⁵⁰⁹ - Attempt to fix segfaults assumed to be caused by future_data instances going out of scope.
- PR #5099⁵¹⁰ - Allow mixing debug and release builds
- PR #5092⁵¹¹ - Replace spirit.qi with x3
- PR #5053⁵¹² - Add P0443r14 executor and a few P1897 algorithms
- PR #5044⁵¹³ - Add performance test in jenkins and reports

⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5172>

⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5171>

⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5170>

⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5166>

⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5165>

⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5164>

⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5160>

⁵⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5158>

⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5155>

⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5153>

⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5152>

⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5150>

⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5144>

⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5125>

⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5117>

⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5099>

⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5092>

⁵¹² <https://github.com/STELLAR-GROUP/hpx/pull/5053>

⁵¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5044>

2.10.3 HPX V1.6.0 (Feb 17, 2021)

General changes

This release continues the focus on C++20 conformance with multiple new algorithms adapted to be C++20 conformant and becoming customization point objects (CPOs). We have also added experimental support for HIP, allowing previous CUDA features to now be compiled with `hipcc` and run on AMD GPUs.

- The following algorithms have been adapted to be C++20 conformant: `adjacent_find`, `includes`, `inplace_merge`, `is_heap`, `is_heap_until`, `is_partitioned`, `is_sorted`, `is_sorted_until`, `merge`, `set_difference`, `set_intersection`, `set_symmetric_difference`, `set_union`.
- Experimental HIP support can be enabled by compiling *HPX* with `hipcc`. All CUDA functionality in *HPX* can now be used with HIP. The HIP functionality is for the time being exposed through the same API as the CUDA functionality, i.e. no changes are required in user code. The CUDA, and now HIP, functionality is in the `hpx::cuda` namespace.
- We have added `partial_sort` based on Francisco Tapia's implementation.
- `hpx::init` and `hpx::start` gained new overloads taking an `hpx::init_params` struct in 1.5.0. All overloads not taking an `hpx::init_params` are now deprecated.
- We have added an experimental `fork_join_executor`. This executor can be used for OpenMP-style fork-join parallelism, where the latency of a parallel region is important for performance.
- The `parallel_executor` now uses a hierarchical spawning scheme for bulk execution, which improves data locality and performance.
- `hpx::dataflow` can now be used with executors that inject additional parameters into the call of the user-provided function.
- We have added experimental support for properties as proposed in P2220⁵¹⁴. Currently the only supported property is the scheduling hint on `parallel_executor`.
- `hpx::util::annotated_function` can now be passed a dynamically generated `std::string`.
- In moving functionality to new namespaces, old names have been deprecated. A deprecation warning will be issued if you are using deprecated functionality, with instructions on how to correct or ignore the warning.
- We have removed all support for C and Fortran from our build system.
- We have further reduced the use of Boost types within *HPX* (`boost::system::error_code` and `boost::detail::spinlock`).
- We have enabled more warnings in our CI builds (unused variables and unused typedefs).

Breaking changes

- `hpxMP` support has been completely removed.
- The verbs `parcelport` has been removed.
- The following compatibility options have been disabled by default: `HPX_WITH_ACTION_BASE_COMPATIBILITY`, `HPX_WITH_REGISTER_THREAD_COMPATIBILITY`, `HPX_WITH_PROMISE_ALIAS_COMPATIBILITY`, `HPX_WITH_UNSCOPED_ENUM_COMPATIBILITY`, `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY`, `HPX_WITH_EMBEDDED_THREAD_POOLS_COMPATIBILITY`, `HPX_WITH_THREAD_POOL_OS_EXECUTOR_COMPATIBILITY`, `HPX_WITH_THREAD_EXECUTORS_COMPATIBILITY`, `HPX_THREAD_AWARE_TIMER_COMPATIBILITY`,

⁵¹⁴ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2220r0.pdf>

HPX_WITH_POOL_EXECUTOR_COMPATIBILITY. Unless noted here, the above functionalities do not come with replacements. Unscoped enumerations have been replaced by scoped enumerations. Previously deprecated unscoped enumerations are disabled by HPX_WITH_UNSCOPE_ENUM_COMPATIBILITY. Newly deprecated unscoped enumerations have been given deprecation warnings and replaced by scoped enumerations. `hpx::promise` has been replaced with `hpx::distributed::promise`. `hpx::program_options` is a drop-in replacement for `boost::program_options`. `hpx::execution::parallel_executor` now has constructors which take a thread pool, covering the use case of `hpx::threads::executors::pool_executor`. A pool can be supplied with `hpx::resource::get_thread_pool`.

Closed issues

- Issue #5148⁵¹⁵ - `runtime_support.hpp` does not work with newer cling
- Issue #5147⁵¹⁶ - Wrong results with parallel reduce
- Issue #5129⁵¹⁷ - Missing specialization for `std::hash<hpx::thread::id>`
- Issue #5126⁵¹⁸ - Use `std::string` for task annotations
- Issue #5115⁵¹⁹ - Don't expect hwloc to always report Cores
- Issue #5113⁵²⁰ - Handle threadmanager exceptions during startup
- Issue #5112⁵²¹ - libatomic problems causing unexpected fails
- Issue #5089⁵²² - Remove non-BSL files
- Issue #5088⁵²³ - Unwrapping problem
- Issue #5087⁵²⁴ - Remove hpxMP support
- Issue #5077⁵²⁵ - PAPI counters are not accessible when HPX is installed
- Issue #5075⁵²⁶ - Make the structs in all `iter_sent.hpp` lower case
- Issue #5067⁵²⁷ - Bug `string_util/split.hpp`
- Issue #5049⁵²⁸ - Change back the hipcc jenkins config to the fury partition on rostam
- Issue #5038⁵²⁹ - Not all examples link in the latest HPX master
- Issue #5035⁵³⁰ - Build with HPX_WITH_EXAMPLES fails
- Issue #5019⁵³¹ - Broken help string for hpx
- Issue #5016⁵³² - `hpx::parallel::fill` fails compiling

⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5148>

⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5147>

⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5129>

⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5126>

⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5115>

⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5113>

⁵²¹ <https://github.com/STELLAR-GROUP/hpx/issues/5112>

⁵²² <https://github.com/STELLAR-GROUP/hpx/issues/5089>

523 <https://github.com/STELLAR-GROUP/hpx/issues/5088>524 <https://github.com/STELLAR-GROUP/hpx/issues/5087>525 <https://github.com/STELLAR-GROUP/hpx/issues/5077>526 <https://github.com/STELLAR-GROUP/hpx/issues/5075>527 <https://github.com/STELLAR-GROUP/hpx/issues/5067>528 <https://github.com/STELLAR-GROUP/hpx/issues/5049>529 <https://github.com/STELLAR-GROUP/hpx/issues/5038>530 <https://github.com/STELLAR-GROUP/hpx/issues/5035>531 <https://github.com/STELLAR-GROUP/hpx/issues/5019>532 <https://github.com/STELLAR-GROUP/hpx/issues/5016>

- Issue #5014⁵³³ - Rename all .cc to .cpp and .hh to .hpp
- Issue #4988⁵³⁴ - MPI is not finalized if running with only one locality
- Issue #4978⁵³⁵ - Change feature test macros to expand to zero/one
- Issue #4949⁵³⁶ - Crash when not enabling TCP parcelport
- Issue #4933⁵³⁷ - Improve test coverage for unused variable warnings etc.
- Issue #4878⁵³⁸ - HPX mpi async might call MPI_FINALIZE before app calls it
- Issue #4127⁵³⁹ - Local runtime entry-points

Closed pull requests

- PR #5178⁵⁴⁰ - Fix parallel remove/remove_copy/transform namespace references in docs
- PR #5169⁵⁴¹ - Attempt to get Piz Daint jenkins setup running after maintenance
- PR #5168⁵⁴² - Remove include of itself
- PR #5167⁵⁴³ - Fixing deprecation warnings that slipped through the net
- PR #5159⁵⁴⁴ - Update APEX tag to 2.3.1
- PR #5154⁵⁴⁵ - Splitting unit tests on circleci to avoid timeouts
- PR #5151⁵⁴⁶ - Use C++20 on clang-newest Jenkins CI configuration
- PR #5149⁵⁴⁷ - Rename 'module' symbols to avoid keyword conflict
- PR #5145⁵⁴⁸ - Adjust handling of CUDA/HIP options in CMake
- PR #5142⁵⁴⁹ - Store annotated_function annotations as std::strings
- PR #5140⁵⁵⁰ - Scheduler mode
- PR #5139⁵⁵¹ - Fix path problem in pre-commit hook, add summary commit line
- PR #5138⁵⁵² - Add program options variable map to resource partitioner init
- PR #5137⁵⁵³ - Remove the use of boost::throw_exception
- PR #5136⁵⁵⁴ - Make sure codespell checks run on CircleCI

⁵³³ <https://github.com/STELLAR-GROUP/hpx/issues/5014>

⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4988>

⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4978>

⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4949>

⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4933>

⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4878>

⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4127>

⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5178>

⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5169>

⁵⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5168>

⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5167>

⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5159>

⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5154>

⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5151>

⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5149>

⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5145>

⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5142>

⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5140>

⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/5139>

⁵⁵² <https://github.com/STELLAR-GROUP/hpx/pull/5138>

⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/5137>

⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5136>

- PR #5132⁵⁵⁵ - Fixing spelling errors
- PR #5131⁵⁵⁶ - Mark `counting_iterator` member functions as `HPX_HOST_DEVICE`
- PR #5130⁵⁵⁷ - Adding specialization for `std::hash<hpx::thread::id>`
- PR #5128⁵⁵⁸ - Fixing environment handling for FreeBSD
- PR #5127⁵⁵⁹ - Fix typo in fibonacci documentation
- PR #5123⁵⁶⁰ - Reduce vector sizes in partial sort benchmarks when running in debug mode
- PR #5122⁵⁶¹ - Making sure exceptions during runtime initialization are correctly reported
- PR #5121⁵⁶² - Working around hwloc limitation on certain platforms
- PR #5120⁵⁶³ - Fixing compatibility warnings in `hpx::transform` implementation
- PR #5119⁵⁶⁴ - Use `sequential_find` and friends from separate detail header
- PR #5116⁵⁶⁵ - Fix compilation with timer pool off
- PR #5114⁵⁶⁶ - Fix 5112 - make sure `libatomic` is used when needed
- PR #5109⁵⁶⁷ - Remove default runtime mode argument from init overload, again
- PR #5108⁵⁶⁸ - Refactor `iter_sent.hpp` to make structs lowercase
- PR #5107⁵⁶⁹ - Relax dataflow internals
- PR #5106⁵⁷⁰ - Change initialization of property CPOs to satisfy older nvcc versions
- PR #5104⁵⁷¹ - Fix regeneration of two files that trigger unnecessary rebuilds
- PR #5103⁵⁷² - Remove default runtime mode argument from start/init overloads
- PR #5102⁵⁷³ - Untie deprecated thread enums from the CMake option
- PR #5101⁵⁷⁴ - Update APEX tag for 1.6.0
- PR #5100⁵⁷⁵ - Bump minimum required Boost version to 1.66 and update CI configurations
- PR #5098⁵⁷⁶ - Minor fixes to public API listing
- PR #5097⁵⁷⁷ - Remove hpxMP support

⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5132>

⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5131>

⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5130>

⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5128>

⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5127>

⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5123>

⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5122>

⁵⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5121>

⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5120>

⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5119>

⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5116>

⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5114>

⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5109>

⁵⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5108>

569 <https://github.com/STELLAR-GROUP/hpx/pull/5107>570 <https://github.com/STELLAR-GROUP/hpx/pull/5106>571 <https://github.com/STELLAR-GROUP/hpx/pull/5104>572 <https://github.com/STELLAR-GROUP/hpx/pull/5103>573 <https://github.com/STELLAR-GROUP/hpx/pull/5102>574 <https://github.com/STELLAR-GROUP/hpx/pull/5101>575 <https://github.com/STELLAR-GROUP/hpx/pull/5100>576 <https://github.com/STELLAR-GROUP/hpx/pull/5098>577 <https://github.com/STELLAR-GROUP/hpx/pull/5097>

- PR #5096⁵⁷⁸ - Remove fractals examples
- PR #5095⁵⁷⁹ - Use all AMD nodes again on rostam
- PR #5094⁵⁸⁰ - Attempt to remove macOS workaround for GH actions environment
- PR #5093⁵⁸¹ - Remove verbs parcelport
- PR #5091⁵⁸² - Avoid moving from lvalues
- PR #5090⁵⁸³ - Adopt C++20 std::endian
- PR #5085⁵⁸⁴ - Update daint CI to use Boost 1.75.0
- PR #5084⁵⁸⁵ - Disable compatibility options for 1.6.0 release
- PR #5083⁵⁸⁶ - Remove duplicated call to the limiting_executor in future_overhead test
- PR #5079⁵⁸⁷ - Add checks to make sure that MPI/CUDA polling is enabled/not disabled too early
- PR #5078⁵⁸⁸ - Add install lib directory to list of component search paths
- PR #5076⁵⁸⁹ - Fix a typo in the jenkins clang-newest cmake config
- PR #5074⁵⁹⁰ - Fixing warnings generated by MSVC
- PR #5073⁵⁹¹ - Allow using noncopyable types with unwrapping
- PR #5072⁵⁹² - Fix is_convertible args in result_types
- PR #5071⁵⁹³ - Fix unused parameters
- PR #5070⁵⁹⁴ - Fix unused variables warnings in hipcc
- PR #5069⁵⁹⁵ - Add support for sentinels to adjacent_find
- PR #5068⁵⁹⁶ - Fix string split function
- PR #5066⁵⁹⁷ - Adapt search to C++20 and Range TS
- PR #5065⁵⁹⁸ - Fix hpx::range::adjacent_find doxygen function signatures
- PR #5064⁵⁹⁹ - Refactor runtime configuration, command line handling, and resource partitioner
- PR #5063⁶⁰⁰ - Limit the device code guards to the distributed parts of the future_overhead bench

⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5096>

⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5095>

⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5094>

⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5093>

⁵⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5091>

⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5090>

⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5085>

⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5084>

⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5083>

⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5079>

⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5078>

⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5076>

⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5074>

⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5073>

⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5072>

⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5071>

⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5070>

⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5069>

⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5068>

⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5066>

⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5065>

⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5064>

⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5063>

- PR #5061⁶⁰¹ - Remove hipcc guards in examples and tests
- PR #5060⁶⁰² - Fix deprecation warnings generated by msvc
- PR #5059⁶⁰³ - Add warning about suspending/resuming the runtime in multi-locality scenarios
- PR #5057⁶⁰⁴ - Fix unused variable warnings
- PR #5056⁶⁰⁵ - Fix hpx::util::get
- PR #5055⁶⁰⁶ - Remove hipcc guards
- PR #5054⁶⁰⁷ - Fix typo
- PR #5051⁶⁰⁸ - Adapt transform to C++20
- PR #5050⁶⁰⁹ - Replace old init overloads in tests and examples
- PR #5048⁶¹⁰ - Limit jenkins hipcc to the reno node
- PR #5047⁶¹¹ - Limit cuda jenkins run to nodes with exclusively Nvidia GPUs
- PR #5046⁶¹² - Convert thread and future enums to class enums
- PR #5043⁶¹³ - Improve hpxrun.py for Phylanx
- PR #5042⁶¹⁴ - Add missing header to partial sort test
- PR #5041⁶¹⁵ - Adding Francisco Tapia's implementation of partial_sort
- PR #5040⁶¹⁶ - Remove generated headers left behind from a previous configuration
- PR #5039⁶¹⁷ - Fix GCC 10 release builds
- PR #5037⁶¹⁸ - Add is_invocable typedefs to top-level hpx namespace and public API list
- PR #5036⁶¹⁹ - Deprecate hpx::util::decay in favor of std::decay
- PR #5034⁶²⁰ - Use versioned container image on CircleCI
- PR #5033⁶²¹ - Implement P2220 properties module
- PR #5032⁶²² - Do codespell comparison only on files changed from common ancestor
- PR #5031⁶²³ - Moving traits files to actions_base

⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5061>

⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5060>

⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5059>

⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5057>

⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5056>

⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5055>

⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5054>

⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5051>

⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5050>

⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5048>

⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5047>

⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/5046>

⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5043>

⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5042>

⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5041>

⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5040>

⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5039>

⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5037>

⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5036>

⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5034>

⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5033>

⁶²² <https://github.com/STELLAR-GROUP/hpx/pull/5032>

⁶²³ <https://github.com/STELLAR-GROUP/hpx/pull/5031>

- PR #5030⁶²⁴ - Add codespell version print in circleci
- PR #5029⁶²⁵ - Work around problems in GitHub actions macOS builder
- PR #5028⁶²⁶ - Moving move files to naming and naming_base
- PR #5027⁶²⁷ - Lessen constraints on certain algorithm arguments
- PR #5025⁶²⁸ - Adapt is_sorted and is_sorted_until to C++20
- PR #5024⁶²⁹ - Moving naming_base to full modules
- PR #5022⁶³⁰ - Remove C language from CMakeLists.txt
- PR #5021⁶³¹ - Warn about unused arguments given to add_hpx_module
- PR #5020⁶³² - Fixing help string
- PR #5018⁶³³ - Update CSCS jenkins configuration to clang 11
- PR #5017⁶³⁴ - Fixing broken backwards compatibility for hpx::parallel::fill
- PR #5015⁶³⁵ - Detect if generated global header conflicts with explicitly listed module headers
- PR #5012⁶³⁶ - Properly reset pointer tracking data in output_archive
- PR #5011⁶³⁷ - Inspect command line tweaks
- PR #5010⁶³⁸ - Creating AGAS module
- PR #5009⁶³⁹ - Replace boost::system::error_code with std::error_code
- PR #5008⁶⁴⁰ - Replace uses of boost::detail::spinlock
- PR #5007⁶⁴¹ - Bump minimal Boost version to 1.65.0
- PR #5006⁶⁴² - Adapt is_partitioned to C++20
- PR #5005⁶⁴³ - Making sure reduce_by_key compiles again
- PR #5004⁶⁴⁴ - Fixing template specializations that make extra archive data types unique across module boundaries
- PR #5003⁶⁴⁵ - Relax dataflow argument constraints
- PR #5001⁶⁴⁶ - Add <random> inspect check

⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5030>

⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5029>

⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5028>

⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5027>

⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5025>

⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5024>

⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5022>

⁶³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5021>

⁶³² <https://github.com/STELLAR-GROUP/hpx/pull/5020>

⁶³³ <https://github.com/STELLAR-GROUP/hpx/pull/5018>

⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5017>

⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5015>

⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5012>

⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5011>

⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5010>

⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5009>

⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5008>

⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5007>

⁶⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5006>

⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5005>

⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5004>

⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5003>

⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5001>

- PR #4999⁶⁴⁷ - Attempt to fix MacOS Github action error
- PR #4997⁶⁴⁸ - Fix unused variable and typedef warnings
- PR #4996⁶⁴⁹ - Adapt adjacent_find to C++20
- PR #4995⁶⁵⁰ - Test all schedulers in cross_pool_injection test except shared_priority_queue_scheduler
- PR #4993⁶⁵¹ - Fix deprecation warnings
- PR #4991⁶⁵² - Avoid unnecessarily including entire modules
- PR #4990⁶⁵³ - Fixing some warnings from HPX complaining about use of obsolete types
- PR #4989⁶⁵⁴ - add a *destroy* trait for ParcelPort plugins
- PR #4986⁶⁵⁵ - Remove serialization to functional module dependency
- PR #4985⁶⁵⁶ - Compatibility header generation
- PR #4980⁶⁵⁷ - Add ranges overloads to for_loop (and variants)
- PR #4979⁶⁵⁸ - Actually enable unity builds on Jenkins
- PR #4977⁶⁵⁹ - Cleaning up debug::print functionalities
- PR #4976⁶⁶⁰ - Remove indirection layer in at_index_impl
- PR #4975⁶⁶¹ - Remove indirection layer in at_index_impl
- PR #4973⁶⁶² - Avoid warnings/errors for older gcc complaining about multi-line comments
- PR #4970⁶⁶³ - Making set algorithms conform to C++20
- PR #4969⁶⁶⁴ - Moving is_execution_policy and friends into namespace hpx
- PR #4968⁶⁶⁵ - Enable deprecation warnings for 1.6.0 and move any functionality to hpx namespace
- PR #4967⁶⁶⁶ - Define deprecation macros conditionally
- PR #4966⁶⁶⁷ - Add clang-format and cmake-format version prints
- PR #4965⁶⁶⁸ - Making is_heap and is_heap_until conforming to C++20
- PR #4964⁶⁶⁹ - Adding parallel make_heap

⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4999>

⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4997>

⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4996>

⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4995>

⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4993>

⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4991>

⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4990>

⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4989>

⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4986>

⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4985>

⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4980>

⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4979>

⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4977>

⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4976>

⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4975>

⁶⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4973>

⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4970>

⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4969>

⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4968>

⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4967>

⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4966>

⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4965>

⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4964>

- PR #4962⁶⁷⁰ - Fix external timer function pointer exports
- PR #4960⁶⁷¹ - Fixing folder names for module tests and examples
- PR #4959⁶⁷² - Adding communications set
- PR #4958⁶⁷³ - Deprecate tuple and timing functionality in `hpx::util`
- PR #4957⁶⁷⁴ - Fixing unity build option for parcelports
- PR #4953⁶⁷⁵ - Fixing MSVC problems after recent restructurings
- PR #4952⁶⁷⁶ - Make `parallel_executor` use `thread_pool_executor` spawning mechanism
- PR #4948⁶⁷⁷ - Clean up old artifacts better and more aggressively on Jenkins
- PR #4947⁶⁷⁸ - Add HIP support for AMD GPUs
- PR #4945⁶⁷⁹ - Enable `HPX_WITH_UNITY_BUILD` option on one of the Jenkins configurations
- PR #4943⁶⁸⁰ - Move public `hpx::parallel::execution` functionality to `hpx::execution`
- PR #4938⁶⁸¹ - Post release cleanup
- PR #4858⁶⁸² - Extending resilience APIs to support distributed invocations
- PR #4744⁶⁸³ - Fork-join executor
- PR #4665⁶⁸⁴ - Implementing sender, receiver, and `operation_state` concepts in terms of P0443r13
- PR #4649⁶⁸⁵ - Split libhpx into multiple libraries
- PR #4642⁶⁸⁶ - Implementing `operation_state` concept in terms of P0443r13
- PR #4640⁶⁸⁷ - Implementing receiver concept in terms of P0443r13
- PR #4622⁶⁸⁸ - Sanitizer fixes

⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4962>

⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4960>

⁶⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4959>

⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4958>

⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4957>

⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4953>

⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4952>

⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4948>

⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4947>

⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4945>

⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4943>

⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4938>

⁶⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4858>

⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4744>

⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4665>

⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4649>

⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4642>

⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4640>

⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4622>

2.10.4 HPX V1.5.1 (Sep 30, 2020)

General changes

This is a patch release. It contains the following changes:

- Remove restriction on suspending runtime with multiple localities, users are now responsible for synchronizing work between localities before suspending.
- Fixes several compilation problems and warnings.
- Adds notes in the documentation explaining how to cite HPX.

Closed issues

- Issue #4971⁶⁸⁹ - Parallel sort fails to compile with C++20
- Issue #4950⁶⁹⁰ - Build with *HPX_WITH_PARCELPORT_ACTION_COUNTERS ON* fails
- Issue #4940⁶⁹¹ - Codespell report for “HPX” (on fossies.org)
- Issue #4937⁶⁹² - Allow suspension of runtime for multiple localities

Closed pull requests

- PR #4982⁶⁹³ - Add page about citing HPX to documentation
- PR #4981⁶⁹⁴ - Adding the missing include
- PR #4974⁶⁹⁵ - Remove leftover format export hack
- PR #4972⁶⁹⁶ - Removing use of get_temporary_buffer and return_temporary_buffer
- PR #4963⁶⁹⁷ - Renaming files to avoid warnings from the vs build system
- PR #4951⁶⁹⁸ - Fixing build if HPX_WITH_PARCELPORT_ACTION_COUNTERS=On
- PR #4946⁶⁹⁹ - Allow suspension on multiple localities
- PR #4944⁷⁰⁰ - Fix typos reported by fossies codespell report
- PR #4941⁷⁰¹ - Adding some explanation to README about how to cite HPX
- PR #4939⁷⁰² - Small changes

⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4971>

⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4950>

⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/4940>

⁶⁹² <https://github.com/STELLAR-GROUP/hpx/issues/4937>

⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4982>

⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4981>

⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4974>

⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4972>

⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4963>

⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4951>

⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4946>

⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4944>

⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4941>

⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4939>

2.10.5 HPX V1.5.0 (Sep 02, 2020)

General changes

The main focus of this release is on APIs and C++20 conformance. We have added many new C++20 features and adapted multiple algorithms to be fully C++20 conformant. As part of the modularization we have begun specifying the public API of *HPX* in terms of headers and functionality, and aligning it more closely to the C++ standard. All non-distributed modules are now in place, along with an experimental option to completely disable distributed features in *HPX*. We have also added experimental asynchronous MPI and CUDA executors. Lastly this release introduces CMake targets for depending projects, performance improvements, and many bug fixes.

- We have added the C++20 features `hpx::jthread` and `hpx::stop_token`. `hpx::condition_variable_any` now exposes new functions supporting `hpx::stop_token`.
- We have added `hpx::stable_sort` based on Francisco Tapia's implementation.
- We have adapted existing synchronization primitives to be fully conformant C++20: `hpx::barrier`, `hpx::latch`, `hpx::counting_semaphore`, and `hpx::binary_semaphore`.
- We have started using customization point objects (CPOs) to make the corresponding algorithms fully conformant to C++20 as well as to make algorithm extension easier for the user. `all_of/any_of/none_of`, `copy`, `count`, `destroy`, `equal`, `fill`, `find`, `for_each`, `generate`, `mismatch`, `move`, `reduce`, `transform_reduce` are using those CPOs (all in namespace `hpx`). We also have adapted their corresponding `hpx::ranges` versions to be conforming to C++20 in this release.
- We have adapted support for `co_await` to C++20, in addition to `hpx::future` it now also supports `hpx::shared_future`. We have also added allocator support for futures returned by `co_return`. It is no longer in the experimental namespace.
- We added serialization support for `std::variant` and `std::tuple`.
- `result_of` and `is_callable` are now deprecated and replaced by `invoke_result` and `is_invocable` to conform to C++20.
- We continued with the modularization, making it easier for us to add the new experimental `HPX_WITH_DISTRIBUTED_RUNTIME` CMake option (see below). A significant amount of headers have been deprecated. We adapted the namespaces and headers we could to be closer to the standard ones (*Public API*). Depending code should still compile, however warnings are now generated instructing to change the include statements accordingly.
- It is now possible to have a basic CUDA support including a helper function to get a future from a CUDA stream and target handling. They are available under the `hpx::cuda::experimental` namespace and they can be enabled with the `-DHPX_WITH_ASYNC_CUDA=ON` CMake option.
- We added a new `hpx::mpi::experimental` namespace for getting futures from an asynchronous MPI call and a new minimal MPI executor `hpx::mpi::experimental::executor`. These can be enabled with the `-DHPX_WITH_ASYNC_MPI=On` CMake option.
- A polymorphic executor has been implemented to reduce compile times as a function accepting executors can potentially be instantiated only once instead of multiple times with different executors. It accepts the function signature as a template argument. It needs to be constructed from any other executor. Please note, that the function signatures that can be scheduled using `then_execute`, `bulk_sync_execute`, `bulk_async_execute` and `bulk_then_execute` are slightly different (See the comment in PR #4514⁷⁰³ for more details).
- The underlying executor of `block_executor` has been updated to a newer one.
- We have added a parameter to `auto_chunk_size` to control the amount of iterations to measure.

⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4514>

- All executor parameter hooks can now be exposed through the executor itself. This will allow to deprecate the `.with()` functionality on execution policies in the future. This is also a first step towards simplifying our executor APIs in preparation for the upcoming C++23 executors (senders/receivers).
- We have moved all of the existing APIs related to resiliency into the namespace `hpx::resiliency::experimental`. Please note this is a breaking change without backwards-compatibility option. We have converted all of those APIs to be based on customization point objects. Two new executors have been added to enable easy integration of the existing resiliency features with other facilities (like the parallel algorithms): `replay_executor` and `replicate_executor`.
- We have added performance counters type information (aggregating, monotonically increasing, average count, average timer, etc.).
- HPX threads are now re-scheduled on the same worker thread they were suspended on to avoid cache misses from moving from one thread to the other. This behavior doesn't prevent the thread from being stolen, however.
- We have added a new configuration option `hpx.exception_verbosity` to allow to control the level of verbosity of the exceptions (3 levels available).
- `broadcast_to`, `broadcast_from`, `scatter_to` and `scatter_from` have been added to the collectives, modernization of `gather_here` and `gather_there` with futures taken by rvalue references. See the breaking change on `all_to_all` in the next section. None of the collectives need supporting macros anymore (e.g. specifying the data types used for a collective operation using `HPX_REGISTER_ALLGATHER` and similar is not needed anymore).
- New API functions have been added: a) to get the number of cores which are idle (`hpx::get_idle_core_count`) and b) returning a bitmask representing the currently idle cores (`hpx::get_idle_core_mask`).
- We have added an experimental option to only enable the local runtime, you can disable the distributed runtime with `HPX_WITH_DISTRIBUTED_RUNTIME=OFF`. You can also enable the local runtime by using the `--hpx:local` runtime option.
- We fixed task annotations for actions.
- The alias `hpx::promise` to `hpx::lcos::promise` is now deprecated. You can use `hpx::lcos::promise` directly instead. `hpx::promise` will refer to the local-only promise in the future.
- We have added a `prepare_checkpoint` API function that calculates the amount of necessary buffer space for a particular set of arguments checkpointed.
- We have added `hpx::upgrade_lock` and `hpx::upgrade_to_unique_lock`, which make `hpx::shared_mutex` (and similar) usable in more flexible ways.
- We have changed the CMake targets exposed to the user, it now includes `HPX::hpx`, `HPX::wrap_main` (int `main` as the first HPX thread of the application, see *Starting the HPX runtime*), `HPX::plugin`, `HPX::component`. The CMake variables `HPX_INCLUDE_DIRS` and `HPX_LIBRARIES` are deprecated and will be removed in a future release, you should now link directly to the `HPX::hpx` CMake target.
- A new example is demonstrating how to create and use a wrapping executor (`quickstart/executor_with_thread_hooks.cpp`)
- A new example is demonstrating how to disable thread stealing during the execution of parallel algorithms (`quickstart/disable_thread_stealing_executor.cpp`)
- We now require for our CMake build system configuration files to be formatted using `cmake-format`.
- We have removed more dependencies on various Boost libraries.
- We have added an experimental option enabling unity builds of HPX using the `-DHPX_WITH_UNITY_BUILD=On` CMake option.

- Many bug fixes.

Breaking changes

- HPX now requires a C++14 capable compiler. We have set the HPX C++ standard automatically to C++14 and if it needs to be set explicitly, it should be specified through the `CMAKE_CXX_STANDARD` setting as mandated by CMake. The `HPX_WITH_CXX*` variables are now deprecated and will be removed in the future.
- Building and using HPX is now supported only when using CMake V3.13 or later, Boost V1.64 or newer, and when compiling with clang V5, gcc V7, or VS2019, or later. Other compilers might still work but have not been tested thoroughly.
- We have added a `hpx::init_params` struct to pass parameters for HPX initialization e.g. the resource partitioner callback to initialize thread pools (*Using the resource partitioner*).
- The `all_to_all` algorithm is renamed to `all_gather`, and the new `all_to_all` algorithm is not compatible with the old one.
- We have moved all of the existing APIs related to resiliency into the namespace `hpx::resiliency::experimental`.

Closed issues

- Issue #4918⁷⁰⁴ - Rename distributed_executors module
- Issue #4900⁷⁰⁵ - Adding JOSS status badge to README
- Issue #4897⁷⁰⁶ - Compiler warning, deprecated header used by HPX itself
- Issue #4886⁷⁰⁷ - A future bound to an action executing on a different locality doesn't capture exception state
- Issue #4880⁷⁰⁸ - Undefined reference to main build error when `HPX_WITH_DYNAMIC_HPX_MAIN=OFF`
- Issue #4877⁷⁰⁹ - `hpx_main` might not able to start hpx runtime properly
- Issue #4850⁷¹⁰ - Issues creating templated component
- Issue #4829⁷¹¹ - Spack package & `HPX_WITH_GENERIC_CONTEXT_COROUTINES`
- Issue #4820⁷¹² - PAPI counters don't work
- Issue #4818⁷¹³ - HPX can't be used with IO pool turned off
- Issue #4816⁷¹⁴ - Build of HPX fails when `find_package(Boost)` is called before `FetchContent_MakeAvailable(hpx)`
- Issue #4813⁷¹⁵ - HPX MPI Future failed
- Issue #4811⁷¹⁶ - Remove `HPX::hpx_no_wrap_main` target before 1.5.0 release

⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4918>

⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4900>

⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4897>

⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4886>

⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4880>

⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4877>

⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4850>

⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/4829>

⁷¹² <https://github.com/STELLAR-GROUP/hpx/issues/4820>

⁷¹³ <https://github.com/STELLAR-GROUP/hpx/issues/4818>

⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4816>

⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4813>

⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4811>

- Issue #4810⁷¹⁷ - In hpx::for_each::invoke_projected the hpx::util::decay is misguided
- Issue #4787⁷¹⁸ - *transform_inclusive_scan* gives incorrect results for non-commutative operator
- Issue #4786⁷¹⁹ - *transform_inclusive_scan* tries to implicitly convert between types, instead of using the provided *conv* function
- Issue #4779⁷²⁰ - HPX build error with GCC 10.1
- Issue #4766⁷²¹ - Move HPX.Compute functionality to experimental namespace
- Issue #4763⁷²² - License file name
- Issue #4758⁷²³ - CMake profiling results
- Issue #4755⁷²⁴ - Building HPX with support for PAPI fails
- Issue #4754⁷²⁵ - CMake cache creation breaks when using HPX with mimalloc
- Issue #4752⁷²⁶ - HPX MPI Future build failed
- Issue #4746⁷²⁷ - Memory leak when using dataflow icw components
- Issue #4731⁷²⁸ - Bug in stencil example, calculation of locality IDs
- Issue #4723⁷²⁹ - Build fail with NETWORKING OFF
- Issue #4720⁷³⁰ - Add compatibility headers for modules that had their module headers implicitly generated in 1.4.1
- Issue #4719⁷³¹ - Undeprecate some module headers
- Issue #4712⁷³² - Rename HPX_MPI_WITH_FUTURES option
- Issue #4709⁷³³ - Make deprecation warnings overridable in dependent projects
- Issue #4691⁷³⁴ - Suggestion to fix and enhance the thread_mapper API
- Issue #4686⁷³⁵ - Fix tutorials examples
- Issue #4685⁷³⁶ - HPX distributed map fails to compile
- Issue #4680⁷³⁷ - Build error with HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- Issue #4679⁷³⁸ - Build error for hpx w/ Apex on Summit

⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4810>

⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4787>

⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4786>

⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4779>

⁷²¹ <https://github.com/STELLAR-GROUP/hpx/issues/4766>

⁷²² <https://github.com/STELLAR-GROUP/hpx/issues/4763>

⁷²³ <https://github.com/STELLAR-GROUP/hpx/issues/4758>

⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4755>

⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4754>

⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4752>

⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4746>

⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4731>

⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4723>

⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4720>

⁷³¹ <https://github.com/STELLAR-GROUP/hpx/issues/4719>

⁷³² <https://github.com/STELLAR-GROUP/hpx/issues/4712>

⁷³³ <https://github.com/STELLAR-GROUP/hpx/issues/4709>

⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4691>

⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4686>

⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4685>

⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4680>

⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4679>

- Issue #4675⁷³⁹ - build error with HPX_WITH_NETWORKING=OFF
- Issue #4674⁷⁴⁰ - Error running Quickstart tests on OS X
- Issue #4662⁷⁴¹ - MPI initialization broken when networking off
- Issue #4652⁷⁴² - How to fix distributed action annotation
- Issue #4650⁷⁴³ - thread descriptions are broken... again
- Issue #4648⁷⁴⁴ - Thread stacksize not properly set
- Issue #4647⁷⁴⁵ - Rename generated collective headers in modules
- Issue #4639⁷⁴⁶ - Update deprecation warnings in compatibility headers to point to collective headers
- Issue #4628⁷⁴⁷ - mpi parcelport totally broken
- Issue #4619⁷⁴⁸ - Fully document hpx_wrap behaviour and targets
- Issue #4612⁷⁴⁹ - Compilation issue with HPX 1.4.1 and 1.4.0
- Issue #4594⁷⁵⁰ - Rename modules
- Issue #4578⁷⁵¹ - Default value for HPX_WITH_THREAD_BACKTRACE_DEPTH
- Issue #4572⁷⁵² - Thread manager should be given a runtime_configuration
- Issue #4571⁷⁵³ - Add high-level documentation to new modules
- Issue #4569⁷⁵⁴ - Annoying warning when compiling - pls suppress or fix it.
- Issue #4555⁷⁵⁵ - HPX_HAVE_THREAD_BACKTRACE_ON_SUSPENSION compilation error
- Issue #4543⁷⁵⁶ - Segfaults in Release builds using *sleep_for*
- Issue #4539⁷⁵⁷ - Compilation Error when HPX_MPI_WITH_FUTURES=ON
- Issue #4537⁷⁵⁸ - Linking issue with libhpx_initd.a
- Issue #4535⁷⁵⁹ - API for checking if pool with a given name exists
- Issue #4523⁷⁶⁰ - Build of PR #4311 (git tag 9955e8e) fails
- Issue #4519⁷⁶¹ - Documentation problem

⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4675>

⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4674>

⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/4662>

⁷⁴² <https://github.com/STELLAR-GROUP/hpx/issues/4652>

⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/4650>

⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4648>

⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4647>

⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4639>

⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4628>

⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4619>

⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4612>

⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4594>

⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/4578>

⁷⁵² <https://github.com/STELLAR-GROUP/hpx/issues/4572>

⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/4571>

⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4569>

⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4555>

⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4543>

⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4539>

⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4537>

⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4535>

⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4523>

⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/4519>

- Issue #4513⁷⁶² - HPXConfig.cmake contains ill-formed paths when library paths use backslashes
- Issue #4507⁷⁶³ - User-polling introduced by MPI futures module should be more generally usable
- Issue #4506⁷⁶⁴ - Make sure force_linking.hpp is not included in main module header
- Issue #4501⁷⁶⁵ - Fix compilation of PAPI tests
- Issue #4497⁷⁶⁶ - Add modules CI checks
- Issue #4489⁷⁶⁷ - Polymorphic executor
- Issue #4476⁷⁶⁸ - Use CMake targets defined by FindBoost
- Issue #4473⁷⁶⁹ - Add vcpkg installation instructions
- Issue #4470⁷⁷⁰ - Adapt hpx::future to C++20 co_await
- Issue #4468⁷⁷¹ - Compile error on Raspberry Pi 4
- Issue #4466⁷⁷² - Compile error on Windows, current stable:
- Issue #4453⁷⁷³ - Installing HPX on fedora with dnf is not adding cmake files
- Issue #4448⁷⁷⁴ - New std::variant serialization broken
- Issue #4438⁷⁷⁵ - Add performance counter flag is monotonically increasing
- Issue #4436⁷⁷⁶ - Build problem: same code build and works with 1.4.0 but it doesn't with 1.4.1
- Issue #4429⁷⁷⁷ - Function descriptions not supported in distributed
- Issue #4423⁷⁷⁸ - --hpx:ini=hpx.lock_detection=0 has no effect
- Issue #4422⁷⁷⁹ - Add performance counter metadata
- Issue #4419⁷⁸⁰ - Weird behavior for --hpx:print-counter-interval with large numbers
- Issue #4401⁷⁸¹ - Create module repository
- Issue #4400⁷⁸² - Command line options conflict related to performance counters
- Issue #4349⁷⁸³ - --hpx:use-process-mask option throw an exception on OS X
- Issue #4345⁷⁸⁴ - Move gh-pages branch out of hpx repo

⁷⁶² <https://github.com/STELLAR-GROUP/hpx/issues/4513>

⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/4507>

⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4506>

⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4501>

⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4497>

⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4489>

⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4476>

⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4473>

⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4470>

771 <https://github.com/STELLAR-GROUP/hpx/issues/4468>772 <https://github.com/STELLAR-GROUP/hpx/issues/4466>773 <https://github.com/STELLAR-GROUP/hpx/issues/4453>774 <https://github.com/STELLAR-GROUP/hpx/issues/4448>775 <https://github.com/STELLAR-GROUP/hpx/issues/4438>776 <https://github.com/STELLAR-GROUP/hpx/issues/4436>777 <https://github.com/STELLAR-GROUP/hpx/issues/4429>778 <https://github.com/STELLAR-GROUP/hpx/issues/4423>779 <https://github.com/STELLAR-GROUP/hpx/issues/4422>780 <https://github.com/STELLAR-GROUP/hpx/issues/4419>781 <https://github.com/STELLAR-GROUP/hpx/issues/4401>782 <https://github.com/STELLAR-GROUP/hpx/issues/4400>783 <https://github.com/STELLAR-GROUP/hpx/issues/4349>784 <https://github.com/STELLAR-GROUP/hpx/issues/4345>

- Issue #4323⁷⁸⁵ - Const-correctness error in assignment operator of compute::vector
- Issue #4318⁷⁸⁶ - ASIO breaks with C++2a concepts
- Issue #4317⁷⁸⁷ - Application runs even if `-hpx:help` is specified
- Issue #4063⁷⁸⁸ - Document hpxcxx compiler wrapper
- Issue #3983⁷⁸⁹ - Implement the C++20 Synchronization Library
- Issue #3696⁷⁹⁰ - C++11 `constexpr` support is now required
- Issue #3623⁷⁹¹ - Modular HPX branch and an alternative project layout
- Issue #2836⁷⁹² - The worst-case time complexity of parallel::sort seems to be O(N^2).

Closed pull requests

- PR #4936⁷⁹³ - Minor documentation fixes part 2
- PR #4935⁷⁹⁴ - Add copyright and license to joss paper file
- PR #4934⁷⁹⁵ - Adding Semicolon in Documentation
- PR #4932⁷⁹⁶ - Fixing compiler warnings
- PR #4931⁷⁹⁷ - Small documentation formatting fixes
- PR #4930⁷⁹⁸ - Documentation Distributed HPX applications localvv with local_vv
- PR #4929⁷⁹⁹ - Add final version of the JOSS paper
- PR #4928⁸⁰⁰ - Add HPX_NODISCARD to enable_user_polling structs
- PR #4926⁸⁰¹ - Rename distributed_executors module to executors_distributed
- PR #4925⁸⁰² - Making transform_reduce conforming to C++20
- PR #4923⁸⁰³ - Don't acquire lock if not needed
- PR #4921⁸⁰⁴ - Update the release notes for the release candidate 3
- PR #4920⁸⁰⁵ - Disable libcds release
- PR #4919⁸⁰⁶ - Make cuda event pool dynamic instead of fixed size

⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4323>

⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4318>

⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4317>

⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4063>

⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3983>

⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3696>

⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3623>

⁷⁹² <https://github.com/STELLAR-GROUP/hpx/issues/2836>

⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4936>

⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4935>

⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4934>

⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4932>

⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4931>

⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4930>

⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4929>

⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4928>

⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4926>

⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4925>

⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4923>

⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4921>

⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4920>

⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4919>

- PR #4917⁸⁰⁷ - Move chrono functionality to hpx::chrono namespace
- PR #4916⁸⁰⁸ - HPX_HAVE_DEPRECATED_WARNINGS needs to be set even when disabled
- PR #4915⁸⁰⁹ - Moving more action related files to actions modules
- PR #4914⁸¹⁰ - Add alias targets with namespaces used for exporting
- PR #4912⁸¹¹ - Aggregate initialize CPOs
- PR #4910⁸¹² - Explicitly specify hwloc root on Jenkins CSCS builds
- PR #4908⁸¹³ - Fix algorithms documentation
- PR #4907⁸¹⁴ - Remove HPX::hpx_no_wrap_main target
- PR #4906⁸¹⁵ - Fixing unused variable warning
- PR #4905⁸¹⁶ - Adding specializations for simple for_loops
- PR #4904⁸¹⁷ - Update boost to 1.74.0 for the newest jenkins configs
- PR #4903⁸¹⁸ - Hide GITHUB_TOKEN environment variables from environment variable output
- PR #4902⁸¹⁹ - Cancel previous pull requests builds before starting a new one with Jenkins
- PR #4901⁸²⁰ - Update public API list with updated algorithms
- PR #4899⁸²¹ - Suggested changes for HPX V1.5 release notes
- PR #4898⁸²² - Minor tweak to hpx::equal implementation
- PR #4896⁸²³ - Making generate() and generate_n conforming to C++20
- PR #4895⁸²⁴ - Update apex tag
- PR #4894⁸²⁵ - Fix exception handling for tasks
- PR #4893⁸²⁶ - Remove last use of std::result_of, removed in C++20
- PR #4892⁸²⁷ - Adding replay_executor and replicate_executor
- PR #4889⁸²⁸ - Restore old behaviour of not requiring linking to hpx_wrap when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4887⁸²⁹ - Making sure remotely thrown (non-hpx) exceptions are properly marshaled back to invocation

⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4917>

⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4916>

⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4915>

⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4914>

⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4912>

⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/4910>

⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4908>

⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4907>

⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4906>

⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4905>

⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4904>

⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4903>

⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4902>

⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4901>

⁸²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4899>

⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/4898>

⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/4896>

⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4895>

⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4894>

⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4893>

⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4892>

⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4889>

⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4887>

site

- PR #4885⁸³⁰ - Adapting hpx::find and friends to C++20
- PR #4884⁸³¹ - Adapting mismatch to C++20
- PR #4883⁸³² - Adapting hpx::equal to be conforming to C++20
- PR #4882⁸³³ - Fixing exception handling for hpx::copy and adding missing tests
- PR #4881⁸³⁴ - Adds different runtime exception when registering thread with the HPX runtime
- PR #4876⁸³⁵ - Adding example demonstrating how to disable thread stealing during the execution of parallel algorithms
- PR #4874⁸³⁶ - Adding non-policy tests to all_of, any_of, and none_of
- PR #4873⁸³⁷ - Set CUDA compute capability on rostam Jenkins builds
- PR #4872⁸³⁸ - Force partitioned vector scan tests to run serially
- PR #4870⁸³⁹ - Making move conforming with C++20
- PR #4869⁸⁴⁰ - Making destroy and destroy_n conforming to C++20
- PR #4868⁸⁴¹ - Fix miscellaneous header problems
- PR #4867⁸⁴² - Add CPOs for for_each
- PR #4865⁸⁴³ - Adapting count and count_if to be conforming to C++20
- PR #4864⁸⁴⁴ - Release notes 1.5.0
- PR #4863⁸⁴⁵ - adding libcds-hpx tag to prepare for hpx 1.5 release
- PR #4862⁸⁴⁶ - Adding version specific deprecation options
- PR #4861⁸⁴⁷ - Limiting executor improvements
- PR #4860⁸⁴⁸ - Making fill and fill_n compatible with C++20
- PR #4859⁸⁴⁹ - Adapting all_of, any_of, and none_of to C++20
- PR #4857⁸⁵⁰ - Improve libCDS integration
- PR #4856⁸⁵¹ - Correct typos in the documentation of the hpx performance counters

⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4885>

⁸³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4884>

⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/4883>

⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/4882>

⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4881>

⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4876>

⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4874>

⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4873>

⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4872>

⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4870>

⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4869>

⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4868>

⁸⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4867>

⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4865>

⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4864>

⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4863>

⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4862>

⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4861>

⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4860>

⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4859>

⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4857>

⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4856>

- PR #4854⁸⁵² - Removing obsolete code
- PR #4853⁸⁵³ - Adding test that derives component from two other components
- PR #4852⁸⁵⁴ - Fix mpi_ring test in distributed mode by ensuring all ranks run hpx_main
- PR #4851⁸⁵⁵ - Converting resiliency APIs to tag_invoke based CPOs
- PR #4849⁸⁵⁶ - Enable use of future_overhead test when DISTRIBUTED_RUNTIME is OFF
- PR #4847⁸⁵⁷ - Fixing ‘error prone’ constructs as reported by Codacy
- PR #4846⁸⁵⁸ - Disable Boost.Aasio concepts support
- PR #4845⁸⁵⁹ - Fix PAPI counters
- PR #4843⁸⁶⁰ - Remove dependency on various Boost headers
- PR #4841⁸⁶¹ - Rearrange public API headers
- PR #4840⁸⁶² - Fixing TSS problems during thread termination
- PR #4839⁸⁶³ - Fix async_cuda build problems when distributed runtime is disabled
- PR #4837⁸⁶⁴ - Restore compatibility for old (now deprecated) copy algorithms
- PR #4836⁸⁶⁵ - Adding CPOs for hpx::reduce
- PR #4835⁸⁶⁶ - Remove *using util::result_of* from namespace hpx
- PR #4834⁸⁶⁷ - Fixing the calculation of the number of idle cores and the corresponding idle masks
- PR #4833⁸⁶⁸ - Allow thread function destructors to yield
- PR #4832⁸⁶⁹ - Fixing assertion in split_gids and memory leaks in 1d_stencil_7
- PR #4831⁸⁷⁰ - Making sure MPI_CXX_COMPILE_FLAGS is interpreted as a sequence of options
- PR #4830⁸⁷¹ - Update documentation on using HPX::wrap_main
- PR #4827⁸⁷² - Update clang-newest configuration to use clang 10
- PR #4826⁸⁷³ - Add Jenkins configuration for rostam
- PR #4825⁸⁷⁴ - Move all CUDA functionality to hpx::cuda::experimental namespace

⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4854>

⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4853>

⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4852>

⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4851>

⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4849>

⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4847>

⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4846>

⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4845>

⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4843>

⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4841>

⁸⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4840>

⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4839>

⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4837>

⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4836>

⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4835>

⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4834>

⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4833>

⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4832>

⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4831>

⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4830>

⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4827>

⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4826>

⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4825>

- PR #4824⁸⁷⁵ - Add support for building master/release branches to Jenkins configuration
- PR #4821⁸⁷⁶ - Implement customization point for hpx::copy and hpx::ranges::copy
- PR #4819⁸⁷⁷ - Allow finding Boost components before finding HPX
- PR #4817⁸⁷⁸ - Adding range version of stable sort
- PR #4815⁸⁷⁹ - Fix a wrong #ifdef for IO/TIMER pools causing build errors
- PR #4814⁸⁸⁰ - Replace hpx::function_nonser with std::function in error module
- PR #4809⁸⁸¹ - Foreach adapt
- PR #4808⁸⁸² - Make internal algorithms functions const
- PR #4807⁸⁸³ - Add Jenkins configuration for running on Piz Daint
- PR #4806⁸⁸⁴ - Update documentation links to new domain name
- PR #4805⁸⁸⁵ - Applying changes that resolve time complexity issues in sort
- PR #4803⁸⁸⁶ - Adding implementation of stable_sort
- PR #4802⁸⁸⁷ - Fix datapar header paths
- PR #4801⁸⁸⁸ - Replace boost::shared_array<T> with std::shared_ptr<T[]> if supported
- PR #4799⁸⁸⁹ - Fixing #include paths in compatibility headers
- PR #4798⁸⁹⁰ - Include the main module header (fixes partially #4488)
- PR #4797⁸⁹¹ - Change cmake targets
- PR #4794⁸⁹² - Removing 128bit integer emulation
- PR #4793⁸⁹³ - Make sure global variable is handled properly
- PR #4792⁸⁹⁴ - Replace enable_if with **HPX_CONCEPT_REQUIREMENTS** and add is_sentinel_for constraint
- PR #4790⁸⁹⁵ - Move deprecation warnings from base template to template specializations for result_of etc. structs
- PR #4789⁸⁹⁶ - Fix hangs during assertion handling and distributed runtime construction
- PR #4788⁸⁹⁷ - Fixing inclusive transform scan algorithm to properly handle initial value

⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4824>

⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4821>

⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4819>

⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4817>

⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4815>

⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4814>

⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4809>

⁸⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4808>

⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4807>

⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4806>

⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4805>

⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4803>

⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4802>

⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4801>

⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4799>

⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4798>

⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4797>

⁸⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4794>

⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4793>

⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4792>

⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4790>

⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4789>

⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4788>

- PR #4785⁸⁹⁸ - Fixing barrier test
- PR #4784⁸⁹⁹ - Fixing deleter argument bindings in serialize_buffer
- PR #4783⁹⁰⁰ - Add coveralls badge
- PR #4782⁹⁰¹ - Make header tests parallel again
- PR #4780⁹⁰² - Remove outdated comment about hpx::stop in documentation
- PR #4776⁹⁰³ - debug print improvements
- PR #4775⁹⁰⁴ - Checkpoint cleanup
- PR #4771⁹⁰⁵ - Fix compilation with HPX_WITH_NETWORKING=OFF
- PR #4767⁹⁰⁶ - Remove all force linking leftovers
- PR #4765⁹⁰⁷ - Fix 1d stencil index calculation
- PR #4764⁹⁰⁸ - Force some tests to run serially
- PR #4762⁹⁰⁹ - Update pointees in compatibility headers
- PR #4761⁹¹⁰ - Fix running and building of execution module tests on CircleCI
- PR #4760⁹¹¹ - Storing hpx_options in global property to speed up summary report
- PR #4759⁹¹² - Reduce memory requirements for our main shared state
- PR #4757⁹¹³ - Fix mimalloc linking on Windows
- PR #4756⁹¹⁴ - Fix compilation issues
- PR #4753⁹¹⁵ - Re-adding API functions that were lost during merges
- PR #4751⁹¹⁶ - Revert “Create coverage reports and upload them to codecov.io”
- PR #4750⁹¹⁷ - Fixing possible race condition during termination detection
- PR #4749⁹¹⁸ - Deprecate result_of and friends
- PR #4748⁹¹⁹ - Create coverage reports and upload them to codecov.io
- PR #4747⁹²⁰ - Changing #include for MPI parcelport

⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4785>

⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4784>

⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4783>

⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4782>

⁹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4780>

⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4776>

⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4775>

⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4771>

⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4767>

⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4765>

⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4764>

⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4762>

⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4761>

⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4760>

⁹¹² <https://github.com/STELLAR-GROUP/hpx/pull/4759>

⁹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4757>

⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4756>

⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4753>

⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4751>

⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4750>

⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4749>

⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4748>

⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4747>

- PR #4745⁹²¹ - Add *is_sentinel_for* trait implementation and test
- PR #4743⁹²² - Fix init_globally example after runtime mode changes
- PR #4742⁹²³ - Update SUPPORT.md
- PR #4741⁹²⁴ - Fixing a warning generated for unity builds with msvc
- PR #4740⁹²⁵ - Rename local_lclos and basic_execution modules
- PR #4739⁹²⁶ - Undeprecate a couple of hpx/modulename.hpp headers
- PR #4738⁹²⁷ - Conditionally test schedulers in thread_stacksize_current test
- PR #4734⁹²⁸ - Fixing a bunch of codacy warnings
- PR #4733⁹²⁹ - Add experimental unity build option to CMake configuration
- PR #4730⁹³⁰ - Fixing compilation problems with unordered map
- PR #4729⁹³¹ - Fix APEX build
- PR #4727⁹³² - Fix missing runtime includes for distributed runtime
- PR #4726⁹³³ - Add more API headers
- PR #4725⁹³⁴ - Add more compatibility headers for deprecated module headers
- PR #4724⁹³⁵ - Fix 4723
- PR #4721⁹³⁶ - Attempt to fixing migration tests
- PR #4717⁹³⁷ - Make the compatibility headers macro conditional
- PR #4716⁹³⁸ - Add hpx/runtime.hpp and hpx/distributed/runtime.hpp API headers
- PR #4714⁹³⁹ - Add hpx/future.hpp header
- PR #4713⁹⁴⁰ - Remove hpx/runtime/threads_fwd.hpp and hpx/util_fwd.hpp
- PR #4711⁹⁴¹ - Make module deprecation warnings overridable
- PR #4710⁹⁴² - Add compatibility headers and other fixes after module header renaming
- PR #4708⁹⁴³ - Add termination handler for parallel algorithms

⁹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4745>

⁹²² <https://github.com/STELLAR-GROUP/hpx/pull/4743>

⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/4742>

⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4741>

⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4740>

⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4739>

⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4738>

⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4734>

⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4733>

⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4730>

⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4729>

⁹³² <https://github.com/STELLAR-GROUP/hpx/pull/4727>

⁹³³ <https://github.com/STELLAR-GROUP/hpx/pull/4726>

⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4725>

⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4724>

⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4721>

⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4717>

⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4716>

⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4714>

⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4713>

⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4711>

⁹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4710>

⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4708>

- PR #4707⁹⁴⁴ - Use hpx::function_nonsr instead of std::function internally
- PR #4706⁹⁴⁵ - Move header file to module
- PR #4705⁹⁴⁶ - Fix incorrect behaviour of cmake-format check
- PR #4704⁹⁴⁷ - Fix resource tests
- PR #4701⁹⁴⁸ - Fix missing includes for future::then specializations
- PR #4700⁹⁴⁹ - Removing obsolete memory component
- PR #4699⁹⁵⁰ - Add short descriptions to modules missing documentation
- PR #4696⁹⁵¹ - Rename generated modules headers
- PR #4693⁹⁵² - Overhauling thread_mapper for public consumption
- PR #4688⁹⁵³ - Fix thread stack size handling
- PR #4687⁹⁵⁴ - Adding all_gather and fixing all_to_all
- PR #4684⁹⁵⁵ - Miscellaneous compilation fixes
- PR #4683⁹⁵⁶ - Fix HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4682⁹⁵⁷ - Fix compilation of pack_traversal_rebind_container.hpp
- PR #4681⁹⁵⁸ - Add missing hpx/execution.hpp includes for future::then
- PR #4678⁹⁵⁹ - Typeless communicator
- PR #4677⁹⁶⁰ - Forcing registry option to be accepted without checks.
- PR #4676⁹⁶¹ - Adding scatter_to/scatter_from collective operations
- PR #4673⁹⁶² - Fix PAPI counters compilation
- PR #4671⁹⁶³ - Deprecate hpx::promise alias to hpx::lcos::promise
- PR #4670⁹⁶⁴ - Explicitly instantiate get_exception
- PR #4667⁹⁶⁵ - Add stopValue in Sentinel struct instead of Iterator
- PR #4666⁹⁶⁶ - Add release build on Windows to GitHub actions

⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4707>

⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4706>

⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4705>

⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4704>

⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4701>

⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4700>

⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4699>

⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4696>

⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4693>

⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4688>

⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4687>

⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4684>

⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4683>

⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4682>

⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4681>

⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4678>

⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4677>

⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4676>

⁹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4673>

⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4671>

⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4670>

⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4667>

⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4666>

- PR #4664⁹⁶⁷ - Creating itt_notify module.
- PR #4663⁹⁶⁸ - Mpi fixes
- PR #4659⁹⁶⁹ - Making sure declarations match definitions in register_locks implementation
- PR #4655⁹⁷⁰ - Fixing task annotations for actions
- PR #4653⁹⁷¹ - Making sure APEX is linked into every application, if needed
- PR #4651⁹⁷² - Update get_function_annotation.hpp
- PR #4646⁹⁷³ - Runtime type
- PR #4645⁹⁷⁴ - Add a few more API headers
- PR #4644⁹⁷⁵ - Fixing support for mpirun (and similar)
- PR #4643⁹⁷⁶ - Fixing the fix for get_idle_core_count() API
- PR #4638⁹⁷⁷ - Remove HPX_API_EXPORT missed in previous cleanup
- PR #4636⁹⁷⁸ - Adding C++20 barrier
- PR #4635⁹⁷⁹ - Adding C++20 latch API
- PR #4634⁹⁸⁰ - Adding C++20 counting semaphore API
- PR #4633⁹⁸¹ - Unify execution parameters customization points
- PR #4632⁹⁸² - Adding missing bulk_sync_execute wrapper to example executor
- PR #4631⁹⁸³ - Updates to documentation; grammar edits.
- PR #4630⁹⁸⁴ - Updates to documentation; moved hyperlink
- PR #4624⁹⁸⁵ - Export set_self_ptr in thread_data.hpp instead of with forward declarations where used
- PR #4623⁹⁸⁶ - Clean up export macros
- PR #4621⁹⁸⁷ - Trigger an error for older boost versions on power architectures
- PR #4617⁹⁸⁸ - Ignore user-set compatibility header options if the module does not have compatibility headers
- PR #4616⁹⁸⁹ - Fix cmake-format warning

⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4664>

⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4663>

⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4659>

⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4655>

⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4653>

⁹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4651>

⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4646>

⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4645>

⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4644>

⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4643>

⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4638>

⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4636>

⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4635>

⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4634>

⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4633>

⁹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4632>

⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4631>

⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4630>

⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4624>

⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4623>

⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4621>

⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4617>

⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4616>

- PR #4615⁹⁹⁰ - Add handler for serializing custom exceptions
- PR #4614⁹⁹¹ - Fix error message when HPX_IGNORE_CMAKE_BUILD_TYPE_COMPATIBILITY=OFF
- PR #4613⁹⁹² - Make partitioner constructor private
- PR #4611⁹⁹³ - Making auto_chunk_size execute the given function using the given executor
- PR #4610⁹⁹⁴ - Making sure the thread-local lock registration data is moving to the core the suspended HPX thread is resumed on
- PR #4609⁹⁹⁵ - Adding an API function that exposes the number of idle cores
- PR #4608⁹⁹⁶ - Fixing moodycamel namespace
- PR #4607⁹⁹⁷ - Moving winsocket initialization to core library
- PR #4606⁹⁹⁸ - Local runtime module etc.
- PR #4604⁹⁹⁹ - Add config_registry module
- PR #4603¹⁰⁰⁰ - Deal with distributed modules in their respective CMakeLists.txt
- PR #4602¹⁰⁰¹ - Small module fixes
- PR #4598¹⁰⁰² - Making sure current_executor and service_executor functions are linked into the core library
- PR #4597¹⁰⁰³ - Adding broadcast_to/broadcast_from to collectives module
- PR #4596¹⁰⁰⁴ - Fix performance regression in block_executor
- PR #4595¹⁰⁰⁵ - Making sure main.cpp is built as a library if HPX_WITH_DYNAMIC_MAIN=OFF
- PR #4592¹⁰⁰⁶ - Futures module
- PR #4591¹⁰⁰⁷ - Adapting co_await support for C++20
- PR #4590¹⁰⁰⁸ - Adding missing exception test for for_loop()
- PR #4587¹⁰⁰⁹ - Move traits headers to hpx/modulename/traits directory
- PR #4586¹⁰¹⁰ - Remove Travis CI config
- PR #4585¹⁰¹¹ - Update macOS test blacklist
- PR #4584¹⁰¹² - Attempting to fix missing symbols in stack trace

990 <https://github.com/STELLAR-GROUP/hpx/pull/4615>

991 <https://github.com/STELLAR-GROUP/hpx/pull/4614>

992 <https://github.com/STELLAR-GROUP/hpx/pull/4613>

993 <https://github.com/STELLAR-GROUP/hpx/pull/4611>

994 <https://github.com/STELLAR-GROUP/hpx/pull/4610>

995 <https://github.com/STELLAR-GROUP/hpx/pull/4609>

996 <https://github.com/STELLAR-GROUP/hpx/pull/4608>

997 <https://github.com/STELLAR-GROUP/hpx/pull/4607>

998 <https://github.com/STELLAR-GROUP/hpx/pull/4606>

999 <https://github.com/STELLAR-GROUP/hpx/pull/4604>

1000 <https://github.com/STELLAR-GROUP/hpx/pull/4603>

1001 <https://github.com/STELLAR-GROUP/hpx/pull/4602>

1002 <https://github.com/STELLAR-GROUP/hpx/pull/4598>

1003 <https://github.com/STELLAR-GROUP/hpx/pull/4597>

1004 <https://github.com/STELLAR-GROUP/hpx/pull/4596>

1005 <https://github.com/STELLAR-GROUP/hpx/pull/4595>

1006 <https://github.com/STELLAR-GROUP/hpx/pull/4592>

1007 <https://github.com/STELLAR-GROUP/hpx/pull/4591>

1008 <https://github.com/STELLAR-GROUP/hpx/pull/4590>

1009 <https://github.com/STELLAR-GROUP/hpx/pull/4587>

1010 <https://github.com/STELLAR-GROUP/hpx/pull/4586>

1011 <https://github.com/STELLAR-GROUP/hpx/pull/4585>

1012 <https://github.com/STELLAR-GROUP/hpx/pull/4584>

- PR #4583¹⁰¹³ - Fixing bad static_cast
- PR #4582¹⁰¹⁴ - Changing download url for Windows prerequisites to circumvent bandwidth limitations
- PR #4581¹⁰¹⁵ - Adding missing using placeholder::_X
- PR #4579¹⁰¹⁶ - Move get_stack_size_name and related functions
- PR #4575¹⁰¹⁷ - Excluding unconditional definition of class backtrace from global header
- PR #4574¹⁰¹⁸ - Changing return type of hardware_concurrency() to unsigned int
- PR #4570¹⁰¹⁹ - Move tests to modules
- PR #4564¹⁰²⁰ - Reshuffle internal targets and add HPX::hpx_no_wrap_main target
- PR #4563¹⁰²¹ - fix CMake option typo
- PR #4562¹⁰²² - Unregister lock earlier to avoid holding it while suspending
- PR #4561¹⁰²³ - Adding test macros supporting custom output stream
- PR #4560¹⁰²⁴ - Making sure hash_any::operator()() is linked into core library
- PR #4559¹⁰²⁵ - Fixing compilation if HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION=On
- PR #4557¹⁰²⁶ - Improve spinlock implementation to perform better in high-contention situations
- PR #4553¹⁰²⁷ - Fix a runtime_ptr problem at shutdown when apex is enabled
- PR #4552¹⁰²⁸ - Add configuration option for making exceptions less noisy
- PR #4551¹⁰²⁹ - Clean up thread creation parameters
- PR #4549¹⁰³⁰ - Test FetchContent build on GitHub actions
- PR #4548¹⁰³¹ - Fix stack size
- PR #4545¹⁰³² - Fix header tests
- PR #4544¹⁰³³ - Fix a typo in sanitizer build
- PR #4541¹⁰³⁴ - Add API to check if a thread pool exists
- PR #4540¹⁰³⁵ - Making sure MPI support is enabled if MPI futures are used but networking is disabled

¹⁰¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4583>

¹⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4582>

¹⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4581>

¹⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4579>

¹⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4575>

¹⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4574>

¹⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4570>

¹⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4564>

¹⁰²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4563>

¹⁰²² <https://github.com/STELLAR-GROUP/hpx/pull/4562>

¹⁰²³ <https://github.com/STELLAR-GROUP/hpx/pull/4561>

¹⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4560>

¹⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4559>

¹⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4557>

¹⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4553>

¹⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4552>

¹⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4551>

¹⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4549>

¹⁰³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4548>

¹⁰³² <https://github.com/STELLAR-GROUP/hpx/pull/4545>

¹⁰³³ <https://github.com/STELLAR-GROUP/hpx/pull/4544>

¹⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4541>

¹⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4540>

- PR #4538¹⁰³⁶ - Move channel documentation examples to examples directory
- PR #4536¹⁰³⁷ - Add generic allocator for execution policies
- PR #4534¹⁰³⁸ - Enable compatibility headers for thread_executors module
- PR #4532¹⁰³⁹ - Fixing broken url in README.rst
- PR #4531¹⁰⁴⁰ - Update scripts
- PR #4530¹⁰⁴¹ - Make sure module API docs show up in correct order
- PR #4529¹⁰⁴² - Adding missing template code to module creation script
- PR #4528¹⁰⁴³ - Make sure version module uses HPX's binary dir, not the parent's
- PR #4527¹⁰⁴⁴ - Creating actions_base and actions module
- PR #4526¹⁰⁴⁵ - Shared state for cv
- PR #4525¹⁰⁴⁶ - Changing sub-name sequencing for experimental namespace
- PR #4524¹⁰⁴⁷ - Add API guarantee notes to API reference documentation
- PR #4522¹⁰⁴⁸ - Enable and fix deprecation warnings in execution module
- PR #4521¹⁰⁴⁹ - Moves more miscellaneous files to modules
- PR #4520¹⁰⁵⁰ - Skip execution customization points when executor is known
- PR #4518¹⁰⁵¹ - Module distributed lcos
- PR #4516¹⁰⁵² - Fix various builds
- PR #4515¹⁰⁵³ - Replace backslashes by slashes in windows paths
- PR #4514¹⁰⁵⁴ - Adding polymorphic_executor
- PR #4512¹⁰⁵⁵ - Adding C++20 jthread and stop_token
- PR #4510¹⁰⁵⁶ - Attempt to fix APEX linking in external packages again
- PR #4508¹⁰⁵⁷ - Only test pull requests (not all branches) with GitHub actions
- PR #4505¹⁰⁵⁸ - Fix duplicate linking in tests (ODR violations)

¹⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4538>

¹⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4536>

¹⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4534>

¹⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4532>

¹⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4531>

¹⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4530>

1042 <https://github.com/STELLAR-GROUP/hpx/pull/4529>1043 <https://github.com/STELLAR-GROUP/hpx/pull/4528>1044 <https://github.com/STELLAR-GROUP/hpx/pull/4527>1045 <https://github.com/STELLAR-GROUP/hpx/pull/4526>1046 <https://github.com/STELLAR-GROUP/hpx/pull/4525>1047 <https://github.com/STELLAR-GROUP/hpx/pull/4524>1048 <https://github.com/STELLAR-GROUP/hpx/pull/4522>1049 <https://github.com/STELLAR-GROUP/hpx/pull/4521>1050 <https://github.com/STELLAR-GROUP/hpx/pull/4520>1051 <https://github.com/STELLAR-GROUP/hpx/pull/4518>1052 <https://github.com/STELLAR-GROUP/hpx/pull/4516>1053 <https://github.com/STELLAR-GROUP/hpx/pull/4515>1054 <https://github.com/STELLAR-GROUP/hpx/pull/4514>1055 <https://github.com/STELLAR-GROUP/hpx/pull/4512>1056 <https://github.com/STELLAR-GROUP/hpx/pull/4510>1057 <https://github.com/STELLAR-GROUP/hpx/pull/4508>1058 <https://github.com/STELLAR-GROUP/hpx/pull/4505>

- PR #4504¹⁰⁵⁹ - Fix C++ standard handling
- PR #4503¹⁰⁶⁰ - Add CMakeLists file check
- PR #4500¹⁰⁶¹ - Fix .clang-format version requirement comment
- PR #4499¹⁰⁶² - Attempting to fix hpx_init linking on macOS
- PR #4498¹⁰⁶³ - Fix compatibility of *pool_executor*
- PR #4496¹⁰⁶⁴ - Removing superfluous SPDX tags
- PR #4494¹⁰⁶⁵ - Module executors
- PR #4493¹⁰⁶⁶ - Pack traversal module
- PR #4492¹⁰⁶⁷ - Update copyright year in documentation
- PR #4491¹⁰⁶⁸ - Add missing current_executor header
- PR #4490¹⁰⁶⁹ - Update GitHub actions configs
- PR #4487¹⁰⁷⁰ - Properly dispatch exceptions thrown from hpx_main to be rethrown from hpx::init/hpx::stop
- PR #4486¹⁰⁷¹ - Fixing an initialization order problem
- PR #4485¹⁰⁷² - Move miscellaneous files to their rightful modules
- PR #4483¹⁰⁷³ - Clean up imported CMake target naming
- PR #4481¹⁰⁷⁴ - Add vcpkg installation instructions
- PR #4479¹⁰⁷⁵ - Add hints to allow to specify MIMALLOC_ROOT
- PR #4478¹⁰⁷⁶ - Async modules
- PR #4475¹⁰⁷⁷ - Fix rp init changes
- PR #4474¹⁰⁷⁸ - Use #pragma once in headers
- PR #4472¹⁰⁷⁹ - Add more descriptive error message when using x86 coroutines on non-x86 platforms
- PR #4467¹⁰⁸⁰ - Add mimalloc find cmake script
- PR #4465¹⁰⁸¹ - Add thread_executors module

¹⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4504>

¹⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4503>

¹⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4500>

¹⁰⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4499>

¹⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4498>

¹⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4496>

¹⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4494>

¹⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4493>

¹⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4492>

¹⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4491>

¹⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4490>

¹⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4487>

¹⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4486>

¹⁰⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4485>

¹⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4483>

¹⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4481>

¹⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4479>

¹⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4478>

¹⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4475>

¹⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4474>

¹⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4472>

¹⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4467>

¹⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4465>

- PR #4464¹⁰⁸² - Include module
- PR #4462¹⁰⁸³ - Merge hpx_init and hpx_wrap into one static library
- PR #4461¹⁰⁸⁴ - Making thread_data test more realistic
- PR #4460¹⁰⁸⁵ - Suppress MPI warnings in version.cpp
- PR #4459¹⁰⁸⁶ - Make sure pkgconfig applications link with hpx_init
- PR #4458¹⁰⁸⁷ - Added example demonstrating how to create and use a wrapping executor
- PR #4457¹⁰⁸⁸ - Fixing execution of thread exit functions
- PR #4456¹⁰⁸⁹ - Move backtrace files to debugging module
- PR #4455¹⁰⁹⁰ - Move deadlock_detection and maintain_queue_wait_times source files into schedulers module
- PR #4450¹⁰⁹¹ - Fixing compilation with std::filesystem enabled
- PR #4449¹⁰⁹² - Fixing build system to actually build variant test
- PR #4447¹⁰⁹³ - This fixes an obsolete #include
- PR #4446¹⁰⁹⁴ - Resume tasks where they were suspended
- PR #4444¹⁰⁹⁵ - Minor CUDA fixes
- PR #4443¹⁰⁹⁶ - Add missing tests to CircleCI config
- PR #4442¹⁰⁹⁷ - Adding a tag to all auto-generated files allowing for tools to visually distinguish those
- PR #4441¹⁰⁹⁸ - Adding performance counter type information
- PR #4440¹⁰⁹⁹ - Fixing MSVC build
- PR #4439¹¹⁰⁰ - Link HPX::plugin and component privately in hpx_setup_target
- PR #4437¹¹⁰¹ - Adding a test that verifies the problem can be solved using a trait specialization
- PR #4434¹¹⁰² - Clean up Boost dependencies and copy string algorithms to new module
- PR #4433¹¹⁰³ - Fixing compilation issues (!) if MPI parcelport is enabled
- PR #4431¹¹⁰⁴ - Ignore warnings about name mangling changing

¹⁰⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4464>

¹⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4462>

¹⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4461>

¹⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4460>

¹⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4459>

¹⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4458>

¹⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4457>

¹⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4456>

¹⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4455>

¹⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4450>

¹⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4449>

¹⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4447>

¹⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4446>

¹⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4444>

¹⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4443>

¹⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4442>

¹⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4441>

¹⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4440>

¹¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4439>

¹¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4437>

¹¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4434>

¹¹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4433>

¹¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4431>

- PR #4430¹¹⁰⁵ - Add performance_counters module
- PR #4428¹¹⁰⁶ - Don't add compatibility headers to module API reference
- PR #4426¹¹⁰⁷ - Add currently failing tests on GitHub actions to blacklist
- PR #4425¹¹⁰⁸ - Clean up and correct minimum required versions
- PR #4424¹¹⁰⁹ - Making sure hpx.lock_detection=0 works as advertised
- PR #4421¹¹¹⁰ - Making sure interval time stops underlying timer thread on termination
- PR #4417¹¹¹¹ - Adding serialization support for std::variant (if available) and std::tuple
- PR #4415¹¹¹² - Partially reverting changes applied by PR 4373
- PR #4414¹¹¹³ - Added documentation for the compiler-wrapper script hpxcxx.in in creating_hpx_projects.rst
- PR #4413¹¹¹⁴ - Merging from V1.4.1 release
- PR #4412¹¹¹⁵ - Making sure to issue a warning if a file specified using --hpx:options-file is not found
- PR #4411¹¹¹⁶ - Make test specific to HPX_WITH_SHARED_PRIORITY_SCHEDULER
- PR #4407¹¹¹⁷ - Adding minimal MPI executor
- PR #4405¹¹¹⁸ - Fix cross pool injection test, use default scheduler as fallback
- PR #4404¹¹¹⁹ - Fix a race condition and clean-up usage of scheduler mode
- PR #4399¹¹²⁰ - Add more threading modules
- PR #4398¹¹²¹ - Add CODEOWNERS file
- PR #4395¹¹²² - Adding a parameter to auto_chunk_size allowing to control the amount of iterations to measure
- PR #4393¹¹²³ - Use appropriate cache-line size defaults for different platforms
- PR #4391¹¹²⁴ - Fixing use of allocator for C++20
- PR #4390¹¹²⁵ - Making --hpx:help behavior consistent
- PR #4388¹¹²⁶ - Change the resource partitioner initialization
- PR #4387¹¹²⁷ - Fix roll_release.sh

¹¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4430>

¹¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4428>

¹¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4426>

¹¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4425>

¹¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4424>

¹¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4421>

1111 <https://github.com/STELLAR-GROUP/hpx/pull/4417>1112 <https://github.com/STELLAR-GROUP/hpx/pull/4415>1113 <https://github.com/STELLAR-GROUP/hpx/pull/4414>1114 <https://github.com/STELLAR-GROUP/hpx/pull/4413>1115 <https://github.com/STELLAR-GROUP/hpx/pull/4412>1116 <https://github.com/STELLAR-GROUP/hpx/pull/4411>1117 <https://github.com/STELLAR-GROUP/hpx/pull/4407>1118 <https://github.com/STELLAR-GROUP/hpx/pull/4405>1119 <https://github.com/STELLAR-GROUP/hpx/pull/4404>1120 <https://github.com/STELLAR-GROUP/hpx/pull/4399>1121 <https://github.com/STELLAR-GROUP/hpx/pull/4398>1122 <https://github.com/STELLAR-GROUP/hpx/pull/4395>1123 <https://github.com/STELLAR-GROUP/hpx/pull/4393>1124 <https://github.com/STELLAR-GROUP/hpx/pull/4391>1125 <https://github.com/STELLAR-GROUP/hpx/pull/4390>1126 <https://github.com/STELLAR-GROUP/hpx/pull/4388>1127 <https://github.com/STELLAR-GROUP/hpx/pull/4387>

- PR #4386¹¹²⁸ - Add warning messages for using thread binding options on macOS
- PR #4385¹¹²⁹ - Cuda futures
- PR #4384¹¹³⁰ - Make enabling dynamic hpx_main on non-Linux systems a configuration error
- PR #4383¹¹³¹ - Use configure_file for HPXCacheVariables.cmake
- PR #4382¹¹³² - Update spellchecking whitelist and fix more typos
- PR #4380¹¹³³ - Add a helper function to get a future from a cuda stream
- PR #4379¹¹³⁴ - Add Windows and macOS CI with GitHub actions
- PR #4378¹¹³⁵ - Change C++ standard handling
- PR #4377¹¹³⁶ - Remove Python scripts
- PR #4374¹¹³⁷ - Adding overload for *hpx::init/hpx::start* for use with resource partitioner
- PR #4373¹¹³⁸ - Adding test that verifies for 4369 to be fixed
- PR #4372¹¹³⁹ - Another attempt at fixing the integral mismatch and conversion warnings
- PR #4370¹¹⁴⁰ - Doc updates quick start
- PR #4368¹¹⁴¹ - Add a whitelist of words for weird spelling suggestions
- PR #4366¹¹⁴² - Suppress or fix clang-tidy-9 warnings
- PR #4365¹¹⁴³ - Removing more Boost dependencies
- PR #4363¹¹⁴⁴ - Update clang-format config file for version 9
- PR #4362¹¹⁴⁵ - Fix indices typo
- PR #4361¹¹⁴⁶ - Boost cleanup
- PR #4360¹¹⁴⁷ - Move plugins
- PR #4358¹¹⁴⁸ - Doc updates; generating documentation. Will likely need heavy editing.
- PR #4356¹¹⁴⁹ - Remove some minor unused and unnecessary Boost includes
- PR #4355¹¹⁵⁰ - Fix spellcheck step in CircleCI config

¹¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4386>

¹¹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4385>

¹¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4384>

¹¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4383>

¹¹³² <https://github.com/STELLAR-GROUP/hpx/pull/4382>

¹¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/4380>

¹¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4379>

¹¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4378>

¹¹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4377>

¹¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4374>

¹¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4373>

¹¹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4372>

¹¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4370>

¹¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4368>

1142 <https://github.com/STELLAR-GROUP/hpx/pull/4366>1143 <https://github.com/STELLAR-GROUP/hpx/pull/4365>1144 <https://github.com/STELLAR-GROUP/hpx/pull/4363>1145 <https://github.com/STELLAR-GROUP/hpx/pull/4362>1146 <https://github.com/STELLAR-GROUP/hpx/pull/4361>1147 <https://github.com/STELLAR-GROUP/hpx/pull/4360>1148 <https://github.com/STELLAR-GROUP/hpx/pull/4358>1149 <https://github.com/STELLAR-GROUP/hpx/pull/4356>1150 <https://github.com/STELLAR-GROUP/hpx/pull/4355>

- PR #4354¹¹⁵¹ - Lightweight utility to hold a pack as members
- PR #4352¹¹⁵² - Minor fixes to the C++ standard detection for MSVC
- PR #4351¹¹⁵³ - Move generated documentation to hpx-docs repo
- PR #4347¹¹⁵⁴ - Add cmake policy - CMP0074
- PR #4346¹¹⁵⁵ - Remove file committed by mistake
- PR #4342¹¹⁵⁶ - Remove HCC and SYCL options from CMakeLists.txt
- PR #4341¹¹⁵⁷ - Fix launch process test with APEX enabled
- PR #4340¹¹⁵⁸ - Testing Cirrus CI
- PR #4339¹¹⁵⁹ - Post 1.4.0 updates
- PR #4338¹¹⁶⁰ - Spelling corrections and CircleCI spell check
- PR #4333¹¹⁶¹ - Flatten bound callables
- PR #4332¹¹⁶² - This is a collection of mostly minor (cleanup) fixes
- PR #4331¹¹⁶³ - This adds the missing tests for async_colocated and async_continue_colocated
- PR #4330¹¹⁶⁴ - Remove HPX.Compute host default_executor
- PR #4328¹¹⁶⁵ - Generate global header for basic_execution module
- PR #4327¹¹⁶⁶ - Use INTERNAL_FLAGS option for all examples and components
- PR #4326¹¹⁶⁷ - Usage of temporary allocator in assignment operator of compute::vector
- PR #4325¹¹⁶⁸ - Use hpx::threads::get_cache_line_size in prefetching.hpp
- PR #4324¹¹⁶⁹ - Enable compatibility headers option for execution module
- PR #4316¹¹⁷⁰ - Add clang format indentppdirectives
- PR #4313¹¹⁷¹ - Introduce index_pack alias to pack of size_t
- PR #4312¹¹⁷² - Fixing compatibility header for pack.hpp
- PR #4311¹¹⁷³ - Dataflow annotations for APEX

¹¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4354>

¹¹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4352>

¹¹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4351>

¹¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4347>

¹¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4346>

¹¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4342>

¹¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4341>

¹¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4340>

¹¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4339>

¹¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4338>

¹¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4333>

¹¹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4332>

¹¹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4331>

¹¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4330>

¹¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4328>

¹¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4327>

¹¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4326>

¹¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4325>

¹¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4324>

¹¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4316>

¹¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4313>

¹¹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4312>

¹¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4311>

- PR #4309¹¹⁷⁴ - Update launching_and_configuring_hpx_applications.rst
- PR #4306¹¹⁷⁵ - Fix schedule hint not being taken from executor
- PR #4305¹¹⁷⁶ - Implementing *hpx::functional::tag_invoke*
- PR #4304¹¹⁷⁷ - Improve pack support utilities
- PR #4303¹¹⁷⁸ - Remove errors module dependency on datastructures
- PR #4301¹¹⁷⁹ - Clean up thread executors
- PR #4294¹¹⁸⁰ - Logging revamp
- PR #4292¹¹⁸¹ - Remove SPDX tag from Boost License file to allow for github to recognize it
- PR #4291¹¹⁸² - Add format support for std::tm
- PR #4290¹¹⁸³ - Simplify compatible tuples check
- PR #4288¹¹⁸⁴ - A lightweight take on boost::lexical_cast
- PR #4287¹¹⁸⁵ - Forking boost::lexical_cast as a new module
- PR #4277¹¹⁸⁶ - MPI_futures
- PR #4270¹¹⁸⁷ - Refactor future implementation
- PR #4265¹¹⁸⁸ - Threading module
- PR #4259¹¹⁸⁹ - Module naming base
- PR #4251¹¹⁹⁰ - Local workrequesting scheduler
- PR #4250¹¹⁹¹ - Inline execution of scoped tasks, if possible
- PR #4247¹¹⁹² - Add execution in module headers
- PR #4246¹¹⁹³ - Expose CMake targets officially
- PR #4239¹¹⁹⁴ - Doc updates miscellaneous (partially completed during Google Season of Docs)
- PR #4233¹¹⁹⁵ - Remove project() from modules + fix CMAKE_SOURCE_DIR issue
- PR #4231¹¹⁹⁶ - Module local lcos

¹¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4309>

¹¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4306>

¹¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4305>

¹¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4304>

¹¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4303>

¹¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4301>

¹¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4294>

¹¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4292>

¹¹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4291>

¹¹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4290>

¹¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4288>

¹¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4287>

¹¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4277>

¹¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4270>

¹¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4265>

¹¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4259>

¹¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4251>

¹¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4250>

1192 <https://github.com/STELLAR-GROUP/hpx/pull/4247>1193 <https://github.com/STELLAR-GROUP/hpx/pull/4246>1194 <https://github.com/STELLAR-GROUP/hpx/pull/4239>1195 <https://github.com/STELLAR-GROUP/hpx/pull/4233>1196 <https://github.com/STELLAR-GROUP/hpx/pull/4231>

- PR #4207¹¹⁹⁷ - Command line handling module
- PR #4206¹¹⁹⁸ - Runtime configuration module
- PR #4141¹¹⁹⁹ - Doc updates examples local to remote (partially completed during Google Season of Docs)
- PR #4091¹²⁰⁰ - Split runtime into local and distributed parts
- PR #4017¹²⁰¹ - Require C++14

2.10.6 HPX V1.4.1 (Feb 12, 2020)

General changes

This is a bugfix release. It contains the following changes:

- Fix compilation issues on Windows, macOS, FreeBSD, and with gcc 10
- Install missing .pdb files on Windows
- Allow running tests using an installed version of *HPX*
- Skip MPI finalization if HPX has not initialized MPI
- Give a hard error when attempting to use IO counters on Windows

Closed issues

- Issue #4320¹²⁰² - HPX 1.4.0 does not compile with gcc 10
- Issue #4336¹²⁰³ - Building HPX 1.4.0 with IO Counters breaks (Windows)
- Issue #4334¹²⁰⁴ - HPX Debug and RelWithDebinfo builds on Windows not installing .pdb files
- Issue #4322¹²⁰⁵ - Undefine VT1 and VT2 after boost includes
- Issue #4314¹²⁰⁶ - Compile error on 1.4.0
- Issue #4307¹²⁰⁷ - ld: error: duplicate symbol: freebsd_environ

¹¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4207>

¹¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4206>

¹¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4141>

¹²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4091>

¹²⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4017>

¹²⁰² <https://github.com/STELLAR-GROUP/hpx/issues/4320>

¹²⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/4336>

¹²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4334>

¹²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4322>

¹²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4314>

¹²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4307>

Closed pull requests

- PR #4376¹²⁰⁸ - Attempt to fix some test build errors on Windows
- PR #4357¹²⁰⁹ - Adding missing #includes to fix gcc V10 linker problems
- PR #4353¹²¹⁰ - Skip MPI_Finalize if MPI_Init is not called from HPX
- PR #4343¹²¹¹ - Give a hard error if IO counters are enabled on non-Linux systems
- PR #4337¹²¹² - Installing pdb files on Windows
- PR #4335¹²¹³ - Adding capability to buildsystem to use an installed version of HPX
- PR #4315¹²¹⁴ - Forcing exported symbols from composable_guard to be linked into core library
- PR #4310¹²¹⁵ - Remove environment handling from exception.cpp

2.10.7 HPX V1.4.0 (January 15, 2020)

General changes

- We have added the collectives all_to_all and all_reduce.
- We have added APIs for resiliency, which allows replication and replay for failed tasks. See the *documentation* for more details.
- Components can now be checkpointed.
- Performance improvements to schedulers and coroutines. A significant change is the addition of stackless coroutines. These are to be used for tasks that do not need to be suspended and can reduce overheads noticeably in applications with short tasks. A stackless coroutine can be created with the new stack size `thread_stacksize_nostack`.
- We have added an implementation of `unique_any`, which is a non-copyable version of `any`.
- The `shared_priority_queue_scheduler` has been improved. It now has lower overheads than the default scheduler in many situations. Unlike the default scheduler it fully supports NUMA scheduling hints. Enable it with the command line option `--hpx:queuing=shared-priority`. This scheduler should still be considered experimental, but its use is encouraged in real applications to help us make it production ready.
- We have added the performance counters `background-receive-duration` and `background-receive-overhead` for inspecting the time and overhead spent on receiving parcels in the background.
- Compilation time has been further improved when `HPX_WITH_NETWORKING=OFF`.
- We no longer require compiled Boost dependencies in certain configurations. This requires at least Boost 1.70, compiling on x86 with GCC 9, clang (libc++) 9, or VS2019 in C++17 mode. The dependency on Boost.Filesystem can explicitly be turned on with `HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY=ON` (it is off by default if the standard library supports `std::filesystem`). Boost.ProgramOptions has been copied into the HPX repository. We have a compatibility layer for users who must explicitly use

¹²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4376>

¹²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4357>

¹²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4353>

¹²¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4343>

¹²¹² <https://github.com/STELLAR-GROUP/hpx/pull/4337>

¹²¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4335>

¹²¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4315>

¹²¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4310>

Boost.ProgramOptions instead of the ProgramOptions provided by HPX. To remove the dependency `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY` must be explicitly set to OFF. This option will be removed in a future release. We have also removed several other header-only dependencies on Boost.

- It is now possible to use the process affinity mask set by tools like numactl and various batch environments with the command line option `--hpx:use-process-mask`. Enabling this option implies `--hpx:ignore-batch-env`.
- It is now possible to create standalone thread pools without starting the runtime. See the `standalone_thread_pool_executor.cpp` test in the execution module for an example.
- Tasks annotated with `hpx::util::annotated_function` now have their correct name when using APEX to generate OTF2 files.
- Cloning of APEX was defective in previous releases (it required manual intervention to check out the correct tag or branch). This has been fixed.
- The option `HPX_WITH_MORE_THAN_64_THREADS` is now ignored and will be removed in a future release. The value is instead derived directly from `HPX_WITH_MAX_CPU_COUNT` option.
- We have deprecated compiling in C++11 mode. The next release will require a C++14 capable compiler.
- We have deprecated support for the Vc library. This option will be replaced with SIMD support from the standard library in a future release.
- We have significantly refactored our CMake setup. This is intended to be a non-breaking change and will allow for using HPX through CMake targets in the future.
- We have continued modularizing the HPX library. In the process we have rearranged many header files into module-specific directories. All moved headers have compatibility headers which forward from the old location to the new location, together with a deprecation warning. The compatibility headers will eventually be removed.
- We now enforce formatting with `clang-format` on the majority of our source files.
- We have added SPDX license tags to all files.
- Many bugfixes.

Breaking changes

- The `HPX_WITH_THREAD_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_UNWRAPPED_COMPATIBILITY` option and the associated compatibility layer has been removed.

Closed issues

- Issue #4282^{[1216](https://github.com/STELLAR-GROUP/hpx/issues/4282)} - Build Issues with Release on Windows
- Issue #4278^{[1217](https://github.com/STELLAR-GROUP/hpx/issues/4278)} - Build Issues with CMake 3.14.4
- Issue #4273^{[1218](https://github.com/STELLAR-GROUP/hpx/issues/4273)} - Clients of HPX 1.4.0-rc2 with APEX ar not linked to libhpx-apex

¹²¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4282>

¹²¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4278>

¹²¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4273>

- Issue #4269¹²¹⁹ - Building HPX 1.4.0-rc2 with support for APEX fails
- Issue #4263¹²²⁰ - Compilation fail on latest master
- Issue #4232¹²²¹ - Configure of HPX project using CMake FetchContent fails
- Issue #4223¹²²² - “Re-using the main() function as the main HPX entry point” doesn’t work
- Issue #4220¹²²³ - HPX won’t compile - error building `resource_partitioner`
- Issue #4215¹²²⁴ - HPX 1.4.0rc1 does not link on s390x
- Issue #4204¹²²⁵ - Trouble compiling HPX with Intel compiler
- Issue #4199¹²²⁶ - Refactor APEX to eliminate circular dependency
- Issue #4187¹²²⁷ - HPX can’t build on OSX
- Issue #4185¹²²⁸ - Simple debug output for development
- Issue #4182¹²²⁹ - @HPX_CONF_PREFIX@ is the empty string
- Issue #4169¹²³⁰ - HPX won’t build with APEX
- Issue #4163¹²³¹ - Add back `HPX_LIBRARIES` and `HPX_INCLUDE_DIRS`
- Issue #4161¹²³² - It should be possible to call `find_package(HPX)` multiple times
- Issue #4155¹²³³ - `get_self_id()` for stackless threads returns `invalid_thread_id`
- Issue #4151¹²³⁴ - build error with MPI code
- Issue #4150¹²³⁵ - hpx won’t build on POWER9 with clang 8
- Issue #4148¹²³⁶ - `cacheline_data` delivers poor performance with C++17 compared to C++14
- Issue #4144¹²³⁷ - target general in `HPX_LIBRARIES` does not exist
- Issue #4134¹²³⁸ - CMake Error when `-DHGX_WITH_HPXMP=ON`
- Issue #4132¹²³⁹ - parallel fill leaves elements unfilled
- Issue #4123¹²⁴⁰ - PAPI performance counters are inaccessible
- Issue #4118¹²⁴¹ - `static_chunk_size` is not obeyed in scan algorithms

¹²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4269>

¹²²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4263>

¹²²¹ <https://github.com/STELLAR-GROUP/hpx/issues/4232>

¹²²² <https://github.com/STELLAR-GROUP/hpx/issues/4223>

¹²²³ <https://github.com/STELLAR-GROUP/hpx/issues/4220>

¹²²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4215>

¹²²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4204>

¹²²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4199>

¹²²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4187>

1228 <https://github.com/STELLAR-GROUP/hpx/issues/4185>1229 <https://github.com/STELLAR-GROUP/hpx/issues/4182>1230 <https://github.com/STELLAR-GROUP/hpx/issues/4169>1231 <https://github.com/STELLAR-GROUP/hpx/issues/4163>1232 <https://github.com/STELLAR-GROUP/hpx/issues/4161>1233 <https://github.com/STELLAR-GROUP/hpx/issues/4155>1234 <https://github.com/STELLAR-GROUP/hpx/issues/4151>1235 <https://github.com/STELLAR-GROUP/hpx/issues/4150>1236 <https://github.com/STELLAR-GROUP/hpx/issues/4148>1237 <https://github.com/STELLAR-GROUP/hpx/issues/4144>1238 <https://github.com/STELLAR-GROUP/hpx/issues/4134>1239 <https://github.com/STELLAR-GROUP/hpx/issues/4132>1240 <https://github.com/STELLAR-GROUP/hpx/issues/4123>1241 <https://github.com/STELLAR-GROUP/hpx/issues/4118>

- Issue #4115¹²⁴² - dependency chaining error with APEX
- Issue #4107¹²⁴³ - Initializing runtime without entry point function and command line arguments
- Issue #4105¹²⁴⁴ - Bug in hpx:bind=numa-balanced
- Issue #4101¹²⁴⁵ - Bound tasks
- Issue #4100¹²⁴⁶ - Add SPDX identifier to all files
- Issue #4085¹²⁴⁷ - hpx_topology library should depend on hwloc
- Issue #4067¹²⁴⁸ - HPX fails to build on macOS
- Issue #4056¹²⁴⁹ - Building without thread manager idle backoff fails
- Issue #4052¹²⁵⁰ - Enforce clang-format style for modules
- Issue #4032¹²⁵¹ - Simple hello world fails to launch correctly
- Issue #4030¹²⁵² - Allow threads to skip context switching
- Issue #4029¹²⁵³ - Add support for mimalloc
- Issue #4005¹²⁵⁴ - Can't link HPX when APEX enabled
- Issue #4002¹²⁵⁵ - Missing header for algorithm module
- Issue #3989¹²⁵⁶ - conversion from long to unsigned int requires a narrowing conversion on MSVC
- Issue #3958¹²⁵⁷ - /statistics/average@ perf counter can't be created
- Issue #3953¹²⁵⁸ - CMake errors from HPX_AddPseudoDependencies
- Issue #3941¹²⁵⁹ - CMake error for APEX install target
- Issue #3940¹²⁶⁰ - Convert pseudo-doxygen function documentation into actual doxygen documentation
- Issue #3935¹²⁶¹ - HPX compiler match too strict?
- Issue #3929¹²⁶² - Buildbot failures on latest HPX stable
- Issue #3912¹²⁶³ - I recommend publishing a version that does not depend on the boost library
- Issue #3890¹²⁶⁴ - hpx.ini not working

¹²⁴² <https://github.com/STELLAR-GROUP/hpx/issues/4115>

¹²⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/4107>

¹²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4105>

¹²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4101>

¹²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4100>

¹²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4085>

1248 <https://github.com/STELLAR-GROUP/hpx/issues/4067>1249 <https://github.com/STELLAR-GROUP/hpx/issues/4056>1250 <https://github.com/STELLAR-GROUP/hpx/issues/4052>1251 <https://github.com/STELLAR-GROUP/hpx/issues/4032>1252 <https://github.com/STELLAR-GROUP/hpx/issues/4030>1253 <https://github.com/STELLAR-GROUP/hpx/issues/4029>1254 <https://github.com/STELLAR-GROUP/hpx/issues/4005>1255 <https://github.com/STELLAR-GROUP/hpx/issues/4002>1256 <https://github.com/STELLAR-GROUP/hpx/issues/3989>1257 <https://github.com/STELLAR-GROUP/hpx/issues/3958>1258 <https://github.com/STELLAR-GROUP/hpx/issues/3953>1259 <https://github.com/STELLAR-GROUP/hpx/issues/3941>1260 <https://github.com/STELLAR-GROUP/hpx/issues/3940>1261 <https://github.com/STELLAR-GROUP/hpx/issues/3935>1262 <https://github.com/STELLAR-GROUP/hpx/issues/3929>1263 <https://github.com/STELLAR-GROUP/hpx/issues/3912>1264 <https://github.com/STELLAR-GROUP/hpx/issues/3890>

- Issue #3883¹²⁶⁵ - cuda compilation fails because of `-faligned-new`
- Issue #3879¹²⁶⁶ - HPX fails to configure with `-DHPX_WITH_TESTS=OFF`
- Issue #3871¹²⁶⁷ - dataflow does not support void allocators
- Issue #3867¹²⁶⁸ - Latest HTML docs placed in wrong directory on GitHub pages
- Issue #3866¹²⁶⁹ - Make sure all tests use `HPX_TEST*` macros and not `HPX_ASSERT`
- Issue #3857¹²⁷⁰ - CMake all-keyword or all-plain for `target_link_libraries`
- Issue #3856¹²⁷¹ - `hpx_setup_target` adds rogue flags
- Issue #3850¹²⁷² - HPX fails to build on POWER8 with Clang7
- Issue #3848¹²⁷³ - Remove `lva` member from `thread_init_data`
- Issue #3838¹²⁷⁴ - `hpx::parallel::count/count_if` failing tests
- Issue #3651¹²⁷⁵ - `hpx::parallel::transform_reduce` with non const reference as lambda parameter
- Issue #3560¹²⁷⁶ - Apex integration with HPX not working properly
- Issue #3322¹²⁷⁷ - No warning when mixing debug/release builds

Closed pull requests

- PR #4300¹²⁷⁸ - Checks for `MPI_Init` being called twice
- PR #4299¹²⁷⁹ - Small CMake fixes
- PR #4298¹²⁸⁰ - Remove extra call to annotate function that messes up traces
- PR #4296¹²⁸¹ - Fixing collectives locking problem
- PR #4295¹²⁸² - Do not check `LICENSE_1_0.txt` for inspect violations
- PR #4293¹²⁸³ - Applying two small changes fixing carious MSVC/Windows problems
- PR #4285¹²⁸⁴ - Delete `apex.hpp`
- PR #4276¹²⁸⁵ - Disable doxygen generation for `hpx/debugging/print.hpp` file
- PR #4275¹²⁸⁶ - Make sure APEX is linked to even when not explicitly referenced

¹²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3883>

¹²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3879>

¹²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3871>

¹²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3867>

¹²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3866>

¹²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3857>

¹²⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/3856>

¹²⁷² <https://github.com/STELLAR-GROUP/hpx/issues/3850>

¹²⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/3848>

¹²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3838>

¹²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3651>

¹²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3560>

¹²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3322>

¹²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4300>

¹²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4299>

¹²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4298>

¹²⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4296>

¹²⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4295>

¹²⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4293>

¹²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4285>

¹²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4276>

¹²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4275>

- PR #4272¹²⁸⁷ - Fix pushing of documentation
- PR #4271¹²⁸⁸ - Updating APEX tag, don't create new task_wrapper on operator= of hpx_thread object
- PR #4268¹²⁸⁹ - Testing for noexcept function specializations in C++11/14 mode
- PR #4267¹²⁹⁰ - Fixing MSVC warning
- PR #4266¹²⁹¹ - Make sure macOS Travis CI fails if build step fails
- PR #4264¹²⁹² - Clean up compatibility header options
- PR #4262¹²⁹³ - Cleanup modules CMakeLists.txt
- PR #4261¹²⁹⁴ - Fixing HPX/APEX linking and dependencies for external projects like Phylanx
- PR #4260¹²⁹⁵ - Fix docs compilation problems
- PR #4258¹²⁹⁶ - Couple of minor changes
- PR #4257¹²⁹⁷ - Fix apex annotation for async dispatch
- PR #4256¹²⁹⁸ - Remove lambdas from assert expressions
- PR #4255¹²⁹⁹ - Ignoring lock in all_to_all and all_reduce
- PR #4254¹³⁰⁰ - Adding action specializations for noexcept functions
- PR #4253¹³⁰¹ - Move partlit.hpp to affinity module
- PR #4252¹³⁰² - Make mismatching build types a hard error in CMake
- PR #4249¹³⁰³ - Scheduler improvement
- PR #4248¹³⁰⁴ - update hpxmp tag to v0.3.0
- PR #4245¹³⁰⁵ - Adding high performance channels
- PR #4244¹³⁰⁶ - Ignore lock in ignore_while_locked_1485 test
- PR #4243¹³⁰⁷ - Fix PAPI command line option documentation
- PR #4242¹³⁰⁸ - Ignore lock in target_distribution_policy
- PR #4241¹³⁰⁹ - Fix start_stop_callbacks test

¹²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4272>

¹²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4271>

¹²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4268>

¹²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4267>

¹²⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4266>

¹²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4264>

1293 <https://github.com/STELLAR-GROUP/hpx/pull/4262>1294 <https://github.com/STELLAR-GROUP/hpx/pull/4261>1295 <https://github.com/STELLAR-GROUP/hpx/pull/4260>1296 <https://github.com/STELLAR-GROUP/hpx/pull/4258>1297 <https://github.com/STELLAR-GROUP/hpx/pull/4257>1298 <https://github.com/STELLAR-GROUP/hpx/pull/4256>1299 <https://github.com/STELLAR-GROUP/hpx/pull/4255>1300 <https://github.com/STELLAR-GROUP/hpx/pull/4254>1301 <https://github.com/STELLAR-GROUP/hpx/pull/4253>1302 <https://github.com/STELLAR-GROUP/hpx/pull/4252>1303 <https://github.com/STELLAR-GROUP/hpx/pull/4249>1304 <https://github.com/STELLAR-GROUP/hpx/pull/4248>1305 <https://github.com/STELLAR-GROUP/hpx/pull/4245>1306 <https://github.com/STELLAR-GROUP/hpx/pull/4244>1307 <https://github.com/STELLAR-GROUP/hpx/pull/4243>1308 <https://github.com/STELLAR-GROUP/hpx/pull/4242>1309 <https://github.com/STELLAR-GROUP/hpx/pull/4241>

- PR #4240¹³¹⁰ - Mostly fix clang CUDA compilation
- PR #4238¹³¹¹ - Google Season of Docs updates to documentation; grammar edits.
- PR #4237¹³¹² - fixing annotated task to use the name, not the desc
- PR #4236¹³¹³ - Move module print summary to modules
- PR #4235¹³¹⁴ - Don't use alignas in cache_{aligned, line}_data
- PR #4234¹³¹⁵ - Add basic overview sentence to all modules
- PR #4230¹³¹⁶ - Add OS X builds to Travis CI
- PR #4229¹³¹⁷ - Remove leftover queue compatibility checks
- PR #4226¹³¹⁸ - Fixing APEX shutdown by explicitly shutting down throttling
- PR #4225¹³¹⁹ - Allow CMAKE_INSTALL_PREFIX to be a relative path
- PR #4224¹³²⁰ - Deprecate verbs parcelport
- PR #4222¹³²¹ - Update register_{thread, work} namespaces
- PR #4221¹³²² - Changing HPX_GCC_VERSION check from 70000 to 70300
- PR #4218¹³²³ - Google Season of Docs updates to documentation; grammar edits.
- PR #4217¹³²⁴ - Google Season of Docs updates to documentation; grammar edits.
- PR #4216¹³²⁵ - Fixing gcc warning on 32bit platforms (integer truncation)
- PR #4214¹³²⁶ - Apex callback refactoring
- PR #4213¹³²⁷ - Clean up allocator checks for dependent projects
- PR #4212¹³²⁸ - Google Season of Docs updates to documentation; grammar edits.
- PR #4211¹³²⁹ - Google Season of Docs updates to documentation; contributing to hpx
- PR #4210¹³³⁰ - Attempting to fix Intel compilation
- PR #4209¹³³¹ - Fix CUDA 10 build
- PR #4205¹³³² - Making sure that differences in CMAKE_BUILD_TYPE are not reported on multi-configuration cmake generators

¹³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4240>

¹³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4238>

¹³¹² <https://github.com/STELLAR-GROUP/hpx/pull/4237>

¹³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4236>

¹³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4235>

¹³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4234>

¹³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4230>

¹³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4229>

¹³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4226>

¹³¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4225>

¹³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4224>

¹³²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4222>

¹³²² <https://github.com/STELLAR-GROUP/hpx/pull/4221>

¹³²³ <https://github.com/STELLAR-GROUP/hpx/pull/4218>

¹³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4217>

¹³²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4216>

¹³²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4214>

¹³²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4213>

¹³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4212>

¹³²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4211>

¹³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4210>

¹³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4209>

¹³³² <https://github.com/STELLAR-GROUP/hpx/pull/4205>

- PR #4203¹³³³ - Deprecate Vc
- PR #4202¹³³⁴ - Fix CUDA configuration
- PR #4200¹³³⁵ - Making sure hpx_wrap is not passed on to linker on non-Linux systems
- PR #4198¹³³⁶ - Fix execution_agent.cpp compilation with GCC 5
- PR #4197¹³³⁷ - Remove deprecated options for 1.4.0 release
- PR #4196¹³³⁸ - minor fixes for building on OSX Darwin
- PR #4195¹³³⁹ - Use full clone on CircleCI for pushing stable tag
- PR #4193¹³⁴⁰ - Add scheduling hints to hello_world_distributed
- PR #4192¹³⁴¹ - Set up CUDA in HPXConfig.cmake
- PR #4191¹³⁴² - Export allocators root variables
- PR #4190¹³⁴³ - Don't use constexpr in thread_data with GCC <= 6
- PR #4189¹³⁴⁴ - Only use quick_exit if available
- PR #4188¹³⁴⁵ - Google Season of Docs updates to documentation; writing single node hpx applications
- PR #4186¹³⁴⁶ - correct vc to cuda in cuda cmake
- PR #4184¹³⁴⁷ - Resetting some cached variables to make sure those are re-filled
- PR #4183¹³⁴⁸ - Fix hpxcxx configuration
- PR #4181¹³⁴⁹ - Rename base libraries var
- PR #4180¹³⁵⁰ - Move header left behind earlier to plugin module
- PR #4179¹³⁵¹ - Moving zip_iterator and transform_iterator to iterator_support module
- PR #4178¹³⁵² - Move checkpointing support to its own module
- PR #4177¹³⁵³ - Small const fix to basic_execution module
- PR #4176¹³⁵⁴ - Add back HPX_LIBRARIES and friends to HPXConfig.cmake
- PR #4175¹³⁵⁵ - Make Vc public and add it to HPXConfig.cmake

¹³³³ <https://github.com/STELLAR-GROUP/hpx/pull/4203>

¹³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4202>

¹³³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4200>

¹³³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4198>

¹³³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4197>

¹³³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4196>

¹³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4195>

¹³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4193>

¹³⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4192>

¹³⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4191>

¹³⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4190>

¹³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4189>

¹³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4188>

¹³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4186>

¹³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4184>

¹³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4183>

¹³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4181>

¹³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4180>

¹³⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4179>

¹³⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4178>

¹³⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4177>

¹³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4176>

¹³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4175>

- PR #4173¹³⁵⁶ - Wait for runtime to be running before returning from hpx::start
- PR #4172¹³⁵⁷ - More protection against shutdown problems in error handling scenarios.
- PR #4171¹³⁵⁸ - Ignore lock in condition_variable::wait
- PR #4170¹³⁵⁹ - Adding APEX dependency to MPI parcelport
- PR #4168¹³⁶⁰ - Adding utility include
- PR #4167¹³⁶¹ - Add a condition to setup the external libraries
- PR #4166¹³⁶² - Add an INTERNAL_FLAGS option to link to hpx_internal_flags
- PR #4165¹³⁶³ - Forward HPX_* cmake cache variables to external projects
- PR #4164¹³⁶⁴ - Affinity and batch environment modules
- PR #4162¹³⁶⁵ - Handle quick_exit
- PR #4160¹³⁶⁶ - Using target_link_libraries for cmake versions >= 3.12
- PR #4159¹³⁶⁷ - Make sure HPX_WITH_NATIVE_TLS is forwarded to dependent projects
- PR #4158¹³⁶⁸ - Adding allocator imported target as a dependency of allocator module
- PR #4157¹³⁶⁹ - Add hpx_memory as a dependency of parcelport plugins
- PR #4156¹³⁷⁰ - Stackless coroutines now can refer to themselves (through get_self() and friends)
- PR #4154¹³⁷¹ - Added CMake policy CMP0060 for HPX applications.
- PR #4153¹³⁷² - add header iomanip to tests and tool
- PR #4152¹³⁷³ - Casting MPI tag value
- PR #4149¹³⁷⁴ - Add back private m_desc member variable in program_options module
- PR #4147¹³⁷⁵ - Resource partitioner and threadmanager modules
- PR #4146¹³⁷⁶ - Google Season of Docs updates to documentation; creating hpx projects
- PR #4145¹³⁷⁷ - Adding basic support for stackless threads
- PR #4143¹³⁷⁸ - Exclude test_client_1950 from all target

¹³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4173>

¹³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4172>

¹³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4171>

¹³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4170>

¹³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4168>

¹³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4167>

¹³⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4166>

¹³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4165>

¹³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4164>

¹³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4162>

¹³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4160>

¹³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4159>

¹³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4158>

¹³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4157>

¹³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4156>

¹³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4154>

¹³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4153>

¹³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4152>

¹³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4149>

¹³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4147>

¹³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4146>

¹³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4145>

¹³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4143>

- PR #4142¹³⁷⁹ - Add a new `thread_pool_executor`
- PR #4140¹³⁸⁰ - Google Season of Docs updates to documentation; why hpx
- PR #4139¹³⁸¹ - Remove runtime includes from coroutines module
- PR #4138¹³⁸² - Forking `boost::intrusive_ptr` and adding it as `hpx::intrusive_ptr`
- PR #4137¹³⁸³ - Fixing TSS destruction
- PR #4136¹³⁸⁴ - HPX.Compute modules
- PR #4133¹³⁸⁵ - Fix `block_executor`
- PR #4131¹³⁸⁶ - Applying fixes based on reports from PVS Studio
- PR #4130¹³⁸⁷ - Adding missing header to build system
- PR #4129¹³⁸⁸ - Fixing compilation if `HPX_WITH_DATAPAR_VC` is enabled
- PR #4128¹³⁸⁹ - Renaming `moveonly_any` to `unique_any`
- PR #4126¹³⁹⁰ - Attempt to fix `basic_any` constructor for gcc 7
- PR #4125¹³⁹¹ - Changing `extra_archive_data` implementation
- PR #4124¹³⁹² - Don't link to Boost.System unless required
- PR #4122¹³⁹³ - Add kernel launch helper utility (+saxpy demo) and merge in octotiger changes
- PR #4121¹³⁹⁴ - Fixing migration test if networking is disabled.
- PR #4120¹³⁹⁵ - Google Season of Docs updates to documentation; hpx build system v1
- PR #4119¹³⁹⁶ - Making sure `chunk_size` and `max_chunk` are actually applied to parallel algorithms if specified
- PR #4117¹³⁹⁷ - Make CircleCI formatting check store diff
- PR #4116¹³⁹⁸ - Fix automatically setting C++ standard
- PR #4114¹³⁹⁹ - Module serialization
- PR #4113¹⁴⁰⁰ - Module datastructures
- PR #4111¹⁴⁰¹ - Fixing performance regression introduced earlier

¹³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4142>

¹³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4140>

¹³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4139>

¹³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4138>

¹³⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4137>

¹³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4136>

¹³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4133>

¹³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4131>

¹³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4130>

¹³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4129>

¹³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4128>

¹³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4126>

¹³⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4125>

¹³⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4124>

¹³⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4122>

¹³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4121>

¹³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4120>

¹³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4119>

¹³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4117>

¹³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4116>

¹³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4114>

¹⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4113>

¹⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4111>

- PR #4110¹⁴⁰² - Adding missing SPDX tags
- PR #4109¹⁴⁰³ - Overload for start without entry point/argv.
- PR #4108¹⁴⁰⁴ - Making sure C++ standard is properly detected and propagated
- PR #4106¹⁴⁰⁵ - use std::round for guaranteed rounding without errors
- PR #4104¹⁴⁰⁶ - Extend scheduler_mode with new work_stealing and task assignment modes
- PR #4103¹⁴⁰⁷ - Add this to lambda capture list
- PR #4102¹⁴⁰⁸ - Add spdx license and check
- PR #4099¹⁴⁰⁹ - Module coroutines
- PR #4098¹⁴¹⁰ - Fix append module path in module CMakeLists template
- PR #4097¹⁴¹¹ - Function tests
- PR #4096¹⁴¹² - Removing return of thread_result_type from functions not needing them
- PR #4095¹⁴¹³ - Stop-gap measure until cmake overhaul is in place
- PR #4094¹⁴¹⁴ - Deprecate HPX_WITH_MORE_THAN_64_THREADS
- PR #4093¹⁴¹⁵ - Fix initialization of global_num_tasks in parallel_executor
- PR #4092¹⁴¹⁶ - Add support for mi-malloc
- PR #4090¹⁴¹⁷ - Execution context
- PR #4089¹⁴¹⁸ - Make counters in coroutines optional
- PR #4087¹⁴¹⁹ - Making hpx::util::any compatible with C++17
- PR #4084¹⁴²⁰ - Making sure destination array for std::transform is properly resized
- PR #4083¹⁴²¹ - Adapting thread_queue_mc to behave even if no 128bit atomics are available
- PR #4082¹⁴²² - Fix compilation on GCC 5
- PR #4081¹⁴²³ - Adding option allowing to force using Boost.FileSystem
- PR #4080¹⁴²⁴ - Updating module dependencies

1402 <https://github.com/STELLAR-GROUP/hpx/pull/4110>

1403 <https://github.com/STELLAR-GROUP/hpx/pull/4109>

1404 <https://github.com/STELLAR-GROUP/hpx/pull/4108>

1405 <https://github.com/STELLAR-GROUP/hpx/pull/4106>

1406 <https://github.com/STELLAR-GROUP/hpx/pull/4104>

1407 <https://github.com/STELLAR-GROUP/hpx/pull/4103>

1408 <https://github.com/STELLAR-GROUP/hpx/pull/4102>

1409 <https://github.com/STELLAR-GROUP/hpx/pull/4099>

1410 <https://github.com/STELLAR-GROUP/hpx/pull/4098>

1411 <https://github.com/STELLAR-GROUP/hpx/pull/4097>

1412 <https://github.com/STELLAR-GROUP/hpx/pull/4096>

1413 <https://github.com/STELLAR-GROUP/hpx/pull/4095>

1414 <https://github.com/STELLAR-GROUP/hpx/pull/4094>

1415 <https://github.com/STELLAR-GROUP/hpx/pull/4093>

1416 <https://github.com/STELLAR-GROUP/hpx/pull/4092>

1417 <https://github.com/STELLAR-GROUP/hpx/pull/4090>

1418 <https://github.com/STELLAR-GROUP/hpx/pull/4089>

1419 <https://github.com/STELLAR-GROUP/hpx/pull/4087>

1420 <https://github.com/STELLAR-GROUP/hpx/pull/4084>

1421 <https://github.com/STELLAR-GROUP/hpx/pull/4083>

1422 <https://github.com/STELLAR-GROUP/hpx/pull/4082>

1423 <https://github.com/STELLAR-GROUP/hpx/pull/4081>

1424 <https://github.com/STELLAR-GROUP/hpx/pull/4080>

- PR #4079¹⁴²⁵ - Add missing tests for iterator_support module
- PR #4078¹⁴²⁶ - Disable parcel-layer if networking is disabled
- PR #4077¹⁴²⁷ - Add missing include that causes build fails
- PR #4076¹⁴²⁸ - Enable compatibility headers for functional module
- PR #4075¹⁴²⁹ - Coroutines module
- PR #4073¹⁴³⁰ - Use configure_file for generated files in modules
- PR #4071¹⁴³¹ - Fixing MPI detection for PMIx
- PR #4070¹⁴³² - Fix macOS builds
- PR #4069¹⁴³³ - Moving more facilities to the collectives module
- PR #4068¹⁴³⁴ - Adding main HPX #include directory to modules
- PR #4066¹⁴³⁵ - Switching the use of message (STATUS "...") to hpx_info
- PR #4065¹⁴³⁶ - Move Boost.Filesystem handling to filesystem module
- PR #4064¹⁴³⁷ - Fix program_options test with older boost versions
- PR #4062¹⁴³⁸ - The cpu_features tool fails to compile on anything but x86 architectures
- PR #4061¹⁴³⁹ - Add clang-format checking step for modules
- PR #4060¹⁴⁴⁰ - Making sure HPX_IDLE_BACKOFF_TIME_MAX is always defined (even if its unused)
- PR #4059¹⁴⁴¹ - Renaming module hpx_parallel_executors into hpx_execution
- PR #4058¹⁴⁴² - Do not build networking tests when networking disabled
- PR #4057¹⁴⁴³ - Printing configuration summary for modules as well
- PR #4055¹⁴⁴⁴ - Google Season of Docs updates to documentation; hpx build systems
- PR #4054¹⁴⁴⁵ - Add troubleshooting section to manual
- PR #4051¹⁴⁴⁶ - Add more variations to future_overhead test
- PR #4050¹⁴⁴⁷ - Creating plugin module

¹⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4079>

¹⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4078>

¹⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4077>

¹⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4076>

¹⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4075>

¹⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4073>

¹⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4071>

¹⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/4070>

¹⁴³³ <https://github.com/STELLAR-GROUP/hpx/pull/4069>

¹⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4068>

¹⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4066>

¹⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4065>

¹⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4064>

¹⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4062>

¹⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4061>

¹⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4060>

¹⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4059>

¹⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4058>

¹⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4057>

¹⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4055>

¹⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4054>

¹⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4051>

¹⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4050>

- PR #4049¹⁴⁴⁸ - Move missing modules tests
- PR #4047¹⁴⁴⁹ - Add boost/filesystem headers to inspect deprecated headers
- PR #4045¹⁴⁵⁰ - Module functional
- PR #4043¹⁴⁵¹ - Fix preconditions and error messages for suspension functions
- PR #4041¹⁴⁵² - Pass HPX_STANDARD on to dependent projects via HPXConfig.cmake
- PR #4040¹⁴⁵³ - Program options module
- PR #4039¹⁴⁵⁴ - Moving non-serializable any (any_nonsr) to datastructures module
- PR #4038¹⁴⁵⁵ - Adding MPark's variant (V1.4.0) to HPX
- PR #4037¹⁴⁵⁶ - Adding resiliency module
- PR #4036¹⁴⁵⁷ - Add C++17 filesystem compatibility header
- PR #4035¹⁴⁵⁸ - Fixing support for mpirun
- PR #4028¹⁴⁵⁹ - CMake to target based directives
- PR #4027¹⁴⁶⁰ - Remove GitLab CI configuration
- PR #4026¹⁴⁶¹ - Threading refactoring
- PR #4025¹⁴⁶² - Refactoring thread queue configuration options
- PR #4024¹⁴⁶³ - Fix padding calculation in cache_aligned_data.hpp
- PR #4023¹⁴⁶⁴ - Fixing Codacy issues
- PR #4022¹⁴⁶⁵ - Make sure process mask option is passed to affinity_data
- PR #4021¹⁴⁶⁶ - Warn about compiling in C++11 mode
- PR #4020¹⁴⁶⁷ - Module concurrency
- PR #4019¹⁴⁶⁸ - Module topology
- PR #4018¹⁴⁶⁹ - Update deprecated header in thread_queue_mc.hpp
- PR #4015¹⁴⁷⁰ - Avoid overwriting artifacts

¹⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4049>

¹⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4047>

¹⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4045>

¹⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4043>

¹⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4041>

¹⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4040>

¹⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4039>

¹⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4038>

¹⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4037>

¹⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4036>

¹⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4035>

¹⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4028>

¹⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4027>

¹⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4026>

¹⁴⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4025>

¹⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4024>

¹⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4023>

¹⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4022>

¹⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4021>

¹⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4020>

¹⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4019>

¹⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4018>

¹⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4015>

- PR #4014¹⁴⁷¹ - Future overheads
- PR #4013¹⁴⁷² - Update URL to test output conversion script
- PR #4012¹⁴⁷³ - Fix CUDA compilation
- PR #4011¹⁴⁷⁴ - Fixing cyclic dependencies between modules
- PR #4010¹⁴⁷⁵ - Ignore stable tag on CircleCI
- PR #4009¹⁴⁷⁶ - Check circular dependencies in a circle ci step
- PR #4008¹⁴⁷⁷ - Extend cache aligned data to handle tuple-like data
- PR #4007¹⁴⁷⁸ - Fixing migration for components that have actions returning a client
- PR #4006¹⁴⁷⁹ - Move is_value_proxy.hpp to algorithms module
- PR #4004¹⁴⁸⁰ - Shorten CTest timeout on CircleCI
- PR #4003¹⁴⁸¹ - Refactoring to remove (internal) dependencies
- PR #4001¹⁴⁸² - Exclude tests from all target
- PR #4000¹⁴⁸³ - Module errors
- PR #3999¹⁴⁸⁴ - Enable support for compatibility headers for logging module
- PR #3998¹⁴⁸⁵ - Add process thread binding option
- PR #3997¹⁴⁸⁶ - Export handle_assert function
- PR #3996¹⁴⁸⁷ - Attempt to solve issue where -latomic does not support 128bit atomics
- PR #3993¹⁴⁸⁸ - Make sure __LINE__ is an unsigned
- PR #3991¹⁴⁸⁹ - Fix dependencies and flags for header tests
- PR #3990¹⁴⁹⁰ - Documentation tags fixes
- PR #3988¹⁴⁹¹ - Adding missing solution folder for format module test
- PR #3987¹⁴⁹² - Move runtime-dependent functions out of command line handling
- PR #3986¹⁴⁹³ - Fix CMake configuration with PAPI on

¹⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4014>

¹⁴⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4013>

¹⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4012>

¹⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4011>

¹⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4010>

¹⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4009>

¹⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4008>

¹⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4007>

¹⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4006>

¹⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4004>

¹⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4003>

¹⁴⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4001>

¹⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4000>

¹⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3999>

¹⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3998>

¹⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3997>

¹⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3996>

¹⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3993>

¹⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3991>

¹⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3990>

¹⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3988>

¹⁴⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3987>

¹⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3986>

- PR #3985¹⁴⁹⁴ - Module timing
- PR #3984¹⁴⁹⁵ - Fix default behaviour of paths in add_hpx_component
- PR #3982¹⁴⁹⁶ - Parallel executors module
- PR #3981¹⁴⁹⁷ - Segmented algorithms module
- PR #3980¹⁴⁹⁸ - Module logging
- PR #3979¹⁴⁹⁹ - Module util
- PR #3978¹⁵⁰⁰ - Fix clang-tidy step on CircleCI
- PR #3977¹⁵⁰¹ - Fixing solution folders for moved components
- PR #3976¹⁵⁰² - Module format
- PR #3975¹⁵⁰³ - Enable deprecation warnings on CircleCI
- PR #3974¹⁵⁰⁴ - Fix typos in documentation
- PR #3973¹⁵⁰⁵ - Fix compilation with GCC 9
- PR #3972¹⁵⁰⁶ - Add condition to clone apex + use of new cmake var APEX_ROOT
- PR #3971¹⁵⁰⁷ - Add testing module
- PR #3968¹⁵⁰⁸ - Remove unneeded file in hardware module
- PR #3967¹⁵⁰⁹ - Remove leftover PIC settings from main CMakeLists.txt
- PR #3966¹⁵¹⁰ - Add missing export option in add_hpx_module
- PR #3965¹⁵¹¹ - Change current_function_helper back to non-constexpr
- PR #3964¹⁵¹² - Fixing merge problems
- PR #3962¹⁵¹³ - Add a trait for std::array for unwrapping
- PR #3961¹⁵¹⁴ - Making hpx::util::tuple<Ts...> and std::tuple<Ts...> convertible
- PR #3960¹⁵¹⁵ - fix compilation with CUDA 10 and GCC 6
- PR #3959¹⁵¹⁶ - Fix C++11 incompatibility

¹⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3985>

¹⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3984>

¹⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3982>

¹⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3981>

¹⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3980>

¹⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3979>

¹⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3978>

¹⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3977>

¹⁵⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3976>

¹⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3975>

¹⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3974>

¹⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3973>

¹⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3972>

¹⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3971>

¹⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3968>

¹⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3967>

¹⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3966>

¹⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3965>

¹⁵¹² <https://github.com/STELLAR-GROUP/hpx/pull/3964>

¹⁵¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3962>

¹⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3961>

¹⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3960>

¹⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3959>

- PR #3957¹⁵¹⁷ - Algorithms module
- PR #3956¹⁵¹⁸ - [HPX_AddModule] Fix lower name var to upper
- PR #3955¹⁵¹⁹ - Fix CMake configuration with examples off and tests on
- PR #3954¹⁵²⁰ - Move components to separate subdirectory in root of repository
- PR #3952¹⁵²¹ - Update papi.cpp
- PR #3951¹⁵²² - Exclude modules header tests from all target
- PR #3950¹⁵²³ - Adding all_reduce facility to collectives module
- PR #3949¹⁵²⁴ - This adds a configuration file that will cause for stale issues to be automatically closed
- PR #3948¹⁵²⁵ - Fixing ALPS environment
- PR #3947¹⁵²⁶ - Add major compiler version check for building hpx as a binary package
- PR #3946¹⁵²⁷ - [Modules] Move the location of the generated headers
- PR #3945¹⁵²⁸ - Simplify tests and examples cmake
- PR #3943¹⁵²⁹ - Remove example module
- PR #3942¹⁵³⁰ - Add NOEXPORT option to add_hpx_{component,library}
- PR #3938¹⁵³¹ - Use https for CDash submissions
- PR #3937¹⁵³² - Add HPX_WITH_BUILD_BINARY_PACKAGE to the compiler check (refs #3935)
- PR #3936¹⁵³³ - Fixing installation of binaries on windows
- PR #3934¹⁵³⁴ - Add set function for sliding_semaphore max_difference
- PR #3933¹⁵³⁵ - Remove cudadevrt from compile/link flags as it breaks downstream projects
- PR #3932¹⁵³⁶ - Fixing 3929
- PR #3931¹⁵³⁷ - Adding all_to_all
- PR #3930¹⁵³⁸ - Add test demonstrating the use of broadcast with component actions
- PR #3928¹⁵³⁹ - fixed number of tasks and number of threads for heterogeneous slurm environments

¹⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3957>

¹⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3956>

¹⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3955>

¹⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3954>

¹⁵²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3952>

¹⁵²² <https://github.com/STELLAR-GROUP/hpx/pull/3951>

¹⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/3950>

¹⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3949>

¹⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3948>

¹⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3947>

¹⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3946>

¹⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3945>

¹⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3943>

¹⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3942>

¹⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3938>

¹⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/3937>

¹⁵³³ <https://github.com/STELLAR-GROUP/hpx/pull/3936>

¹⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3934>

¹⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3933>

¹⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3932>

¹⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3931>

¹⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3930>

¹⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3928>

- PR #3927¹⁵⁴⁰ - Moving Cache module's tests into separate solution folder
- PR #3926¹⁵⁴¹ - Move unit tests to cache module
- PR #3925¹⁵⁴² - Move version check to config module
- PR #3924¹⁵⁴³ - Add schedule hint executor parameters
- PR #3923¹⁵⁴⁴ - Allow aligning objects bigger than the cache line size
- PR #3922¹⁵⁴⁵ - Add Windows builds with Travis CI
- PR #3921¹⁵⁴⁶ - Add ccls cache directory to gitignore
- PR #3920¹⁵⁴⁷ - Fix git_external fetching of tags
- PR #3905¹⁵⁴⁸ - Correct rostambod url. Fix typo in doc
- PR #3904¹⁵⁴⁹ - Fix bug in context_base.hpp
- PR #3903¹⁵⁵⁰ - Adding new performance counters
- PR #3902¹⁵⁵¹ - Add add_hpx_module function
- PR #3901¹⁵⁵² - Factoring out container remapping into a separate trait
- PR #3900¹⁵⁵³ - Making sure errors during command line processing are properly reported and will not cause assertions
- PR #3899¹⁵⁵⁴ - Remove old compatibility bases from make_action
- PR #3898¹⁵⁵⁵ - Make parameter size be of type size_t
- PR #3897¹⁵⁵⁶ - Making sure all tests are disabled if HPX_WITH_TESTS=OFF
- PR #3895¹⁵⁵⁷ - Add documentation for annotated_function
- PR #3894¹⁵⁵⁸ - Working around VS2019 problem with make_action
- PR #3892¹⁵⁵⁹ - Avoid MSVC compatibility warning in internal allocator
- PR #3891¹⁵⁶⁰ - Removal of the default intel config include
- PR #3888¹⁵⁶¹ - Fix async_customization dataflow example and Clarify what's being tested
- PR #3887¹⁵⁶² - Add Doxygen documentation

1540 <https://github.com/STELLAR-GROUP/hpx/pull/3927>

1541 <https://github.com/STELLAR-GROUP/hpx/pull/3926>

1542 <https://github.com/STELLAR-GROUP/hpx/pull/3925>

1543 <https://github.com/STELLAR-GROUP/hpx/pull/3924>

1544 <https://github.com/STELLAR-GROUP/hpx/pull/3923>

1545 <https://github.com/STELLAR-GROUP/hpx/pull/3922>

1546 <https://github.com/STELLAR-GROUP/hpx/pull/3921>

1547 <https://github.com/STELLAR-GROUP/hpx/pull/3920>

1548 <https://github.com/STELLAR-GROUP/hpx/pull/3905>

1549 <https://github.com/STELLAR-GROUP/hpx/pull/3904>

1550 <https://github.com/STELLAR-GROUP/hpx/pull/3903>

1551 <https://github.com/STELLAR-GROUP/hpx/pull/3902>

1552 <https://github.com/STELLAR-GROUP/hpx/pull/3901>

1553 <https://github.com/STELLAR-GROUP/hpx/pull/3900>

1554 <https://github.com/STELLAR-GROUP/hpx/pull/3899>

1555 <https://github.com/STELLAR-GROUP/hpx/pull/3898>

1556 <https://github.com/STELLAR-GROUP/hpx/pull/3897>

1557 <https://github.com/STELLAR-GROUP/hpx/pull/3895>

1558 <https://github.com/STELLAR-GROUP/hpx/pull/3894>

1559 <https://github.com/STELLAR-GROUP/hpx/pull/3892>

1560 <https://github.com/STELLAR-GROUP/hpx/pull/3891>

1561 <https://github.com/STELLAR-GROUP/hpx/pull/3888>

1562 <https://github.com/STELLAR-GROUP/hpx/pull/3887>

- PR #3882¹⁵⁶³ - Minor docs fixes
- PR #3880¹⁵⁶⁴ - Updating APEX version tag
- PR #3878¹⁵⁶⁵ - Making sure symbols are properly exported from modules (needed for Windows/MacOS)
- PR #3877¹⁵⁶⁶ - Documentation
- PR #3876¹⁵⁶⁷ - Module hardware
- PR #3875¹⁵⁶⁸ - Converted typedefs in actions submodule to using directives
- PR #3874¹⁵⁶⁹ - Allow one to suppress target keywords in hpx_setup_target for backwards compatibility
- PR #3873¹⁵⁷⁰ - Add scripts to create releases and generate lists of PRs and issues
- PR #3872¹⁵⁷¹ - Fix latest HTML docs location
- PR #3870¹⁵⁷² - Module cache
- PR #3869¹⁵⁷³ - Post 1.3.0 version bumps
- PR #3868¹⁵⁷⁴ - Replace the macro HPX_ASSERT by HPX_TEST in tests
- PR #3845¹⁵⁷⁵ - Assertion module
- PR #3839¹⁵⁷⁶ - Make tuple serialization non-intrusive
- PR #3832¹⁵⁷⁷ - Config module
- PR #3799¹⁵⁷⁸ - Remove compat namespace and its contents
- PR #3701¹⁵⁷⁹ - MoodyCamel lockfree
- PR #3496¹⁵⁸⁰ - Disabling MPI's (deprecated) C++ interface
- PR #3192¹⁵⁸¹ - Move type info into hpx::debug namespace and add print helper functions
- PR #3159¹⁵⁸² - Support Checkpointing Components

¹⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3882>

¹⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3880>

¹⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3878>

¹⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3877>

¹⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3876>

¹⁵⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3875>

¹⁵⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3874>

¹⁵⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3873>

¹⁵⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3872>

¹⁵⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3870>

¹⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3869>

¹⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3868>

¹⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3845>

¹⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3839>

¹⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3832>

¹⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3799>

¹⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3701>

¹⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3496>

¹⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3192>

¹⁵⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3159>

2.10.8 HPX V1.3.0 (May 23, 2019)

General changes

- Performance improvements: the schedulers have significantly reduced overheads from removing false sharing and the parallel executor has been updated to create fewer futures.
- HPX now defaults to not turning on networking when running on one locality. This means that you can run multiple instances on the same system without adding command line options.
- Multiple issues reported by Clang sanitizers have been fixed.
- We have added (back) single-page HTML documentation and PDF documentation.
- We have started modularizing the HPX library. This is useful both for developers and users. In the long term users will be able to consume only parts of the HPX libraries if they do not require all the functionality that HPX currently provides.
- We have added an implementation of `function_ref`.
- The `barrier` and `latch` classes have gained a few additional member functions.

Breaking changes

- Executable and library targets are now created without the `_exe` and `_lib` suffix respectively. For example, the target `1d_stencil_1_exe` is now simply called `1d_stencil_1`.
- We have removed the following deprecated functionality: `queue`, `scoped_unlock`, and support for input iterators in algorithms.
- We have turned off the compatibility layer for `unwrapped` by default. The functionality will be removed in the next release. The option can still be turned on using the `CMake`¹⁵⁸³ option `HPX_WITH_UNWRAPPED_SUPPORT`. Likewise, `inclusive_scan` compatibility overloads have been turned off by default. They can still be turned on with `HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY`.
- The minimum compiler and dependency versions have been updated. We now support GCC from version 5 onwards, Clang from version 4 onwards, and Boost from version 1.61.0 onwards.
- The headers for preprocessor macros have moved as a result of the functionality being moved to a separate module. The old headers are deprecated and will be removed in a future version of HPX. You can turn off the warnings by setting `HPX_PREPROCESSOR_WITH_DEPRECATED_WARNINGS=OFF` or turn off the compatibility headers completely with `HPX_PREPROCESSOR_WITH_COMPATIBILITY_HEADERS=OFF`.

Closed issues

- Issue #3863¹⁵⁸⁴ - shouldn't “-faligned-new” be a usage requirement?
- Issue #3841¹⁵⁸⁵ - Build error with msvc 19 caused by SFINAE and C++17
- Issue #3836¹⁵⁸⁶ - master branch does not build with idle rate counters enabled
- Issue #3819¹⁵⁸⁷ - Add debug suffix to modules built in debug mode
- Issue #3817¹⁵⁸⁸ - `HPX_INCLUDE_DIRS` contains non-existent directory

¹⁵⁸³ <https://www.cmake.org>

¹⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3863>

¹⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3841>

¹⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3836>

¹⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3819>

¹⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3817>

- Issue #3810¹⁵⁸⁹ - Source groups are not created for files in modules
- Issue #3805¹⁵⁹⁰ - HPX won't compile with `-DHPX_WITH_APEX=TRUE`
- Issue #3792¹⁵⁹¹ - Barrier Hangs When Locality Zero not included
- Issue #3778¹⁵⁹² - Replace `throw()` with `noexcept`
- Issue #3763¹⁵⁹³ - configurable sort limit per task
- Issue #3758¹⁵⁹⁴ - dataflow doesn't convert `future<future<T>>` to `future<T>`
- Issue #3757¹⁵⁹⁵ - When compiling undefined reference to `hpx::hpx_check_version_1_2` HPX V1.2.1, Ubuntu 18.04.01 Server Edition
- Issue #3753¹⁵⁹⁶ - `--hpx:list-counters=full` crashes
- Issue #3746¹⁵⁹⁷ - Detection of MPI with pmix
- Issue #3744¹⁵⁹⁸ - Separate spinlock from same cacheline as internal data for all LCOs
- Issue #3743¹⁵⁹⁹ - `hpxcxx`'s shebang doesn't specify the python version
- Issue #3738¹⁶⁰⁰ - Unable to debug parcelport on a single node
- Issue #3735¹⁶⁰¹ - Latest master: Can't compile in MSVC
- Issue #3731¹⁶⁰² - `util::bound` seems broken on Clang with older libstdc++
- Issue #3724¹⁶⁰³ - Allow to pre-set command line options through environment
- Issue #3723¹⁶⁰⁴ - examples/resource_partitioner build issue on master branch / ubuntu 18
- Issue #3721¹⁶⁰⁵ - faced a building error
- Issue #3720¹⁶⁰⁶ - Hello World example fails to link
- Issue #3719¹⁶⁰⁷ - pkg-config produces invalid output: `-l-pthread`
- Issue #3718¹⁶⁰⁸ - Please make the python executable configurable through cmake
- Issue #3717¹⁶⁰⁹ - interested to contribute to the organisation
- Issue #3699¹⁶¹⁰ - Remove 'HPX runtime' executable
- Issue #3698¹⁶¹¹ - Ignore all locks while handling asserts

¹⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3810>

¹⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3805>

¹⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3792>

¹⁵⁹² <https://github.com/STELLAR-GROUP/hpx/issues/3778>

¹⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/3763>

¹⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3758>

¹⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3757>

¹⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3753>

¹⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3746>

¹⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3744>

¹⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3743>

¹⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3738>

¹⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/3735>

¹⁶⁰² <https://github.com/STELLAR-GROUP/hpx/issues/3731>

¹⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/3724>

¹⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3723>

¹⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3721>

¹⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3720>

¹⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3719>

¹⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3718>

¹⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3717>

¹⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3699>

¹⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3698>

- Issue #3689¹⁶¹² - Incorrect and inconsistent website structure <http://stellar.cct.lsu.edu/downloads/>
- Issue #3681¹⁶¹³ - Broken links on <http://stellar.cct.lsu.edu/2015/05/hpx-archives-now-on-gmane/>
- Issue #3676¹⁶¹⁴ - HPX master built from source, cmake fails to link main.cpp example in docs
- Issue #3673¹⁶¹⁵ - HPX build fails with `std::atomic` missing error
- Issue #3670¹⁶¹⁶ - Generate PDF again from documentation (with Sphinx)
- Issue #3643¹⁶¹⁷ - Warnings when compiling HPX 1.2.1 with gcc 9
- Issue #3641¹⁶¹⁸ - Trouble with using ranges-v3 and `hpx::parallel::reduce`
- Issue #3639¹⁶¹⁹ - `util::unwrapping` does not work well with member functions
- Issue #3634¹⁶²⁰ - The build fails if `shared_future<>::then` is called with a thread executor
- Issue #3622¹⁶²¹ - VTune Amplifier 2019 not working with `use_itt_notify=1`
- Issue #3616¹⁶²² - HPX Fails to Build with CUDA 10
- Issue #3612¹⁶²³ - False sharing of scheduling counters
- Issue #3609¹⁶²⁴ - `executor_parameters timeout` with gcc <= 7 and Debug mode
- Issue #3601¹⁶²⁵ - Misleading error message on power pc for `rdtsc` and `rdtscp`
- Issue #3598¹⁶²⁶ - Build of some examples fails when using Vc
- Issue #3594¹⁶²⁷ - Error: The number of OS threads requested (20) does not match the number of threads to bind (12): `HPX(bad_parameter)`
- Issue #3592¹⁶²⁸ - Undefined Reference Error
- Issue #3589¹⁶²⁹ - include could not find load file: `HPX_Utils.cmake`
- Issue #3587¹⁶³⁰ - HPX won't compile on POWER8 with Clang 7
- Issue #3583¹⁶³¹ - Fedora and openSUSE instructions missing on "Distribution Packages" page
- Issue #3578¹⁶³² - Build error when configuring with `HPX_HAVE_ALGORITHM_INPUT_ITERATOR_SUPPORT=ON`
- Issue #3575¹⁶³³ - Merge openSUSE reproducible patch
- Issue #3570¹⁶³⁴ - Update HPX to work with the latest VC version

¹⁶¹² <https://github.com/STELLAR-GROUP/hpx/issues/3689>

¹⁶¹³ <https://github.com/STELLAR-GROUP/hpx/issues/3681>

¹⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3676>

¹⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3673>

¹⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3670>

¹⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3643>

¹⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3641>

¹⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3639>

¹⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3634>

¹⁶²¹ <https://github.com/STELLAR-GROUP/hpx/issues/3622>

¹⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/3616>

¹⁶²³ <https://github.com/STELLAR-GROUP/hpx/issues/3612>

¹⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3609>

¹⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3601>

¹⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3598>

¹⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3594>

¹⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3592>

¹⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3589>

¹⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3587>

¹⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/3583>

¹⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/3578>

¹⁶³³ <https://github.com/STELLAR-GROUP/hpx/issues/3575>

¹⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3570>

- Issue #3567¹⁶³⁵ - Build succeed and make failed for hpx : cout
- Issue #3565¹⁶³⁶ - Polymorphic simple component destructor not getting called
- Issue #3559¹⁶³⁷ - 1.2.0 is missing from download page
- Issue #3554¹⁶³⁸ - Clang 6.0 warning of hiding overloaded virtual function
- Issue #3510¹⁶³⁹ - Build on ppc64 fails
- Issue #3482¹⁶⁴⁰ - Improve error message when HPX_WITH_MAX_CPU_COUNT is too low for given system
- Issue #3453¹⁶⁴¹ - Two HPX applications can't run at the same time.
- Issue #3452¹⁶⁴² - Scaling issue on the change to 2 NUMA domains
- Issue #3442¹⁶⁴³ - HPX set_difference, set_intersection failure cases
- Issue #3437¹⁶⁴⁴ - Ensure parent_task pointer when child task is created and child/parent are on same locality
- Issue #3255¹⁶⁴⁵ - Suspension with lock for --hpx:list-component-types
- Issue #3034¹⁶⁴⁶ - Use C++17 structured bindings for serialization
- Issue #2999¹⁶⁴⁷ - Change thread scheduling use of size_t for thread indexing

Closed pull requests

- PR #3865¹⁶⁴⁸ - adds hpx_target_compile_option_if_available
- PR #3864¹⁶⁴⁹ - Helper functions that are useful in numa binding and testing of allocator
- PR #3862¹⁶⁵⁰ - Temporary fix to local_dataflow_boost_small_vector test
- PR #3860¹⁶⁵¹ - Add cache line padding to intermediate results in for loop reduction
- PR #3859¹⁶⁵² - Remove HPX_TLL_PUBLIC and HPX_TLL_PRIVATE from CMake files
- PR #3858¹⁶⁵³ - Add compile flags and definitions to modules
- PR #3851¹⁶⁵⁴ - update hpxmp release tag to v0.2.0
- PR #3849¹⁶⁵⁵ - Correct BOOST_ROOT variable name in quick start guide
- PR #3847¹⁶⁵⁶ - Fix attach_debugger configuration option

¹⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3567>

¹⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3565>

¹⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3559>

¹⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3554>

¹⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3510>

¹⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3482>

¹⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/3453>

¹⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/3452>

¹⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/3442>

¹⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3437>

¹⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3255>

¹⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3034>

¹⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2999>

¹⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3865>

¹⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3864>

¹⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3862>

¹⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3860>

¹⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3859>

¹⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3858>

¹⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3851>

¹⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3849>

¹⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3847>

- PR #3846¹⁶⁵⁷ - Add tests for libs header tests
- PR #3844¹⁶⁵⁸ - Fixing source_groups in preprocessor module to properly handle compatibility headers
- PR #3843¹⁶⁵⁹ - This fixes the launch_process/launched_process pair of tests
- PR #3842¹⁶⁶⁰ - Fix macro call with ITTNOTIFY enabled
- PR #3840¹⁶⁶¹ - Fixing SLURM environment parsing
- PR #3837¹⁶⁶² - Fixing misplaced #endif
- PR #3835¹⁶⁶³ - make all latch members protected for consistency
- PR #3834¹⁶⁶⁴ - Disable transpose_block numa example on CircleCI
- PR #3833¹⁶⁶⁵ - make latch **counter** protected for deriving latch in hpxmp
- PR #3831¹⁶⁶⁶ - Fix CircleCI config for modules
- PR #3830¹⁶⁶⁷ - minor fix: option HPX_WITH_TEST was not working correctly
- PR #3828¹⁶⁶⁸ - Avoid for binaries that depend on HPX to directly link against internal modules
- PR #3827¹⁶⁶⁹ - Adding shortcut for `hpx::get_ptr<>(sync, id)` for a local, non-migratable objects
- PR #3826¹⁶⁷⁰ - Fix and update modules documentation
- PR #3825¹⁶⁷¹ - Updating default APEX version to 2.1.3 with HPX
- PR #3823¹⁶⁷² - Fix pkgconfig libs handling
- PR #3822¹⁶⁷³ - Change includes in `hpx_wrap.cpp` to more specific includes
- PR #3821¹⁶⁷⁴ - Disable barrier_3792 test when networking is disabled
- PR #3820¹⁶⁷⁵ - Assorted CMake fixes
- PR #3815¹⁶⁷⁶ - Removing left-over debug output
- PR #3814¹⁶⁷⁷ - Allow setting default scheduler mode via the configuration database
- PR #3813¹⁶⁷⁸ - Make the deprecation warnings issued by the old pp headers optional
- PR #3812¹⁶⁷⁹ - Windows requires to handle symlinks to directories differently from those linking files

¹⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3846>

¹⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3844>

¹⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3843>

¹⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3842>

¹⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3840>

¹⁶⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3837>

¹⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3835>

¹⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3834>

¹⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3833>

¹⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3831>

¹⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3830>

¹⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3828>

¹⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3827>

¹⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3826>

¹⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3825>

¹⁶⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3823>

¹⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3822>

¹⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3821>

¹⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3820>

¹⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3815>

¹⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3814>

¹⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3813>

¹⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3812>

- PR #3811¹⁶⁸⁰ - Clean up PP module and library skeleton
- PR #3806¹⁶⁸¹ - Moving include path configuration to before APEX
- PR #3804¹⁶⁸² - Fix latch
- PR #3803¹⁶⁸³ - Update hpxcxx to look at lib64 and use python3
- PR #3802¹⁶⁸⁴ - Numa binding allocator
- PR #3801¹⁶⁸⁵ - Remove duplicated includes
- PR #3800¹⁶⁸⁶ - Attempt to fix Posix context switching after lazy init changes
- PR #3798¹⁶⁸⁷ - count and count_if accepts different iterator types
- PR #3797¹⁶⁸⁸ - Adding a couple of `override` keywords to overloaded virtual functions
- PR #3796¹⁶⁸⁹ - Re-enable testing all schedulers in `shutdown_suspended_test`
- PR #3795¹⁶⁹⁰ - Change `std::terminate` to `std::abort` in SIGSEGV handler
- PR #3794¹⁶⁹¹ - Fixing #3792
- PR #3793¹⁶⁹² - Extending `migrate_polymorphic_component` unit test
- PR #3791¹⁶⁹³ - Change `throw()` to `noexcept`
- PR #3790¹⁶⁹⁴ - Remove deprecated options for 1.3.0 release
- PR #3789¹⁶⁹⁵ - Remove Boost filesystem compatibility header
- PR #3788¹⁶⁹⁶ - Disabled even more spots that should not execute if networking is disabled
- PR #3787¹⁶⁹⁷ - Bump minimal boost supported version to 1.61.0
- PR #3786¹⁶⁹⁸ - Bump minimum required versions for 1.3.0 release
- PR #3785¹⁶⁹⁹ - Explicitly set number of jobs for all ninja invocations on CircleCI
- PR #3784¹⁷⁰⁰ - Fix leak and address sanitizer problems
- PR #3783¹⁷⁰¹ - Disabled even more spots that should not execute if networking is disabled
- PR #3782¹⁷⁰² - Cherry-picked tuple and `thread_init_data` fixes from #3701

¹⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3811>

¹⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3806>

¹⁶⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3804>

¹⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3803>

¹⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3802>

¹⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3801>

¹⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3800>

¹⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3798>

¹⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3797>

¹⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3796>

¹⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3795>

¹⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3794>

¹⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3793>

¹⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3791>

¹⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3790>

¹⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3789>

¹⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3788>

¹⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3787>

¹⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3786>

¹⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3785>

¹⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3784>

¹⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3783>

¹⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3782>

- PR #3781¹⁷⁰³ - Fix generic context coroutines after lazy stack allocation changes
- PR #3780¹⁷⁰⁴ - Rename hello world examples
- PR #3776¹⁷⁰⁵ - Sort algorithms now use the supplied chunker to determine the required minimal chunk size
- PR #3775¹⁷⁰⁶ - Disable Boost auto-linking
- PR #3774¹⁷⁰⁷ - Tag and push stable builds
- PR #3773¹⁷⁰⁸ - Enable migration of polymorphic components
- PR #3771¹⁷⁰⁹ - Fix link to stackoverflow in documentation
- PR #3770¹⁷¹⁰ - Replacing constexpr if in brace-serialization code
- PR #3769¹⁷¹¹ - Fix SIGSEGV handler
- PR #3768¹⁷¹² - Adding flags to scheduler allowing to control thread stealing and idle back-off
- PR #3767¹⁷¹³ - Fix help formatting in hpxrun.py
- PR #3765¹⁷¹⁴ - Fix a couple of bugs in the thread test
- PR #3764¹⁷¹⁵ - Workaround for SFINAE regression in msvc14.2
- PR #3762¹⁷¹⁶ - Prevent MSVC from prematurely instantiating things
- PR #3761¹⁷¹⁷ - Update python scripts to work with python 3
- PR #3760¹⁷¹⁸ - Fix callable vtable for GCC4.9
- PR #3759¹⁷¹⁹ - Rename PAGE_SIZE to PAGE_SIZE_ because AppleClang
- PR #3755¹⁷²⁰ - Making sure locks are not held during suspension
- PR #3754¹⁷²¹ - Disable more code if networking is not available/not enabled
- PR #3752¹⁷²² - Move util::format implementation to source file
- PR #3751¹⁷²³ - Fixing problems with lcos::barrier and iostreams
- PR #3750¹⁷²⁴ - Change error message to take into account use_guard_page setting
- PR #3749¹⁷²⁵ - Fix lifetime problem in run_as_hpx_thread

¹⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3781>

¹⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3780>

¹⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3776>

¹⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3775>

¹⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3774>

¹⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3773>

¹⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3771>

¹⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3770>

¹⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3769>

¹⁷¹² <https://github.com/STELLAR-GROUP/hpx/pull/3768>

¹⁷¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3767>

¹⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3765>

¹⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3764>

¹⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3762>

¹⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3761>

¹⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3760>

¹⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3759>

¹⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3755>

¹⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3754>

¹⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/3752>

¹⁷²³ <https://github.com/STELLAR-GROUP/hpx/pull/3751>

¹⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3750>

¹⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3749>

- PR #3748¹⁷²⁶ - Fixed unusable behavior of the clang code analyzer.
- PR #3747¹⁷²⁷ - Added PMIX_RANK to the defaults of HPX_WITH_PARCELPORT_MPI_ENV.
- PR #3745¹⁷²⁸ - Introduced cache_aligned_data and cache_line_data helper structure
- PR #3742¹⁷²⁹ - Remove more unused functionality from util/logging
- PR #3740¹⁷³⁰ - Fix includes in partitioned vector tests
- PR #3739¹⁷³¹ - More fixes to make sure that std::flush really flushes all output
- PR #3737¹⁷³² - Fix potential shutdown problems
- PR #3736¹⁷³³ - Fix guided_pool_executor after dataflow changes caused compilation fail
- PR #3734¹⁷³⁴ - Limiting executor
- PR #3732¹⁷³⁵ - More constrained bound constructors
- PR #3730¹⁷³⁶ - Attempt to fix deadlocks during component loading
- PR #3729¹⁷³⁷ - Add latch member function count_up and reset, requested by hpxMP
- PR #3728¹⁷³⁸ - Send even empty buffers on hpx::endl and hpx::flush
- PR #3727¹⁷³⁹ - Adding example demonstrating how to customize the memory management for a component
- PR #3726¹⁷⁴⁰ - Adding support for passing command line options through the HPX_COMMANDLINE_OPTIONS environment variable
- PR #3722¹⁷⁴¹ - Document known broken OpenMPI builds
- PR #3716¹⁷⁴² - Add barrier reset function, requested by hpxMP for reusing barrier
- PR #3715¹⁷⁴³ - More work on functions and vtables
- PR #3714¹⁷⁴⁴ - Generate single-page HTML, PDF, manpage from documentation
- PR #3713¹⁷⁴⁵ - Updating default APEX version to 2.1.2
- PR #3712¹⁷⁴⁶ - Update release procedure
- PR #3710¹⁷⁴⁷ - Fix the C++11 build, after #3704
- PR #3709¹⁷⁴⁸ - Move some component_registry functionality to source file

¹⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3748>

¹⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3747>

¹⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3745>

¹⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3742>

¹⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3740>

¹⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3739>

¹⁷³² <https://github.com/STELLAR-GROUP/hpx/pull/3737>

¹⁷³³ <https://github.com/STELLAR-GROUP/hpx/pull/3736>

¹⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3734>

¹⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3732>

¹⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3730>

¹⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3729>

¹⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3728>

¹⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3727>

¹⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3726>

¹⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3722>

¹⁷⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3716>

¹⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3715>

¹⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3714>

¹⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3713>

¹⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3712>

¹⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3710>

¹⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3709>

- PR #3708¹⁷⁴⁹ - Ignore all locks while handling assertions
- PR #3707¹⁷⁵⁰ - Remove obsolete hpx runtime executable
- PR #3705¹⁷⁵¹ - Fix and simplify `make_ready_future` overload sets
- PR #3704¹⁷⁵² - Reduce use of binders
- PR #3703¹⁷⁵³ - Ini
- PR #3702¹⁷⁵⁴ - Fixing CUDA compiler errors
- PR #3700¹⁷⁵⁵ - Added `barrier::increment` function to increase total number of thread
- PR #3697¹⁷⁵⁶ - One more attempt to fix migration...
- PR #3694¹⁷⁵⁷ - Fixing component migration
- PR #3693¹⁷⁵⁸ - Print thread state when getting disallowed value in `set_thread_state`
- PR #3692¹⁷⁵⁹ - Only disable `constexpr` with clang-cuda, not nvcc+gcc
- PR #3691¹⁷⁶⁰ - Link with `libspsc++` if needed for `thread_local`
- PR #3690¹⁷⁶¹ - Remove thousands separators in `set_operations_3442` to comply with C++11
- PR #3688¹⁷⁶² - Decouple serialization from function vtables
- PR #3687¹⁷⁶³ - Fix a couple of test failures
- PR #3686¹⁷⁶⁴ - Make sure tests.unit.build are run after install on CircleCI
- PR #3685¹⁷⁶⁵ - Revise quickstart CMakeLists.txt explanation
- PR #3684¹⁷⁶⁶ - Provide concept emulation for Ranges-TS concepts
- PR #3683¹⁷⁶⁷ - Ignore uninitialized chunks
- PR #3682¹⁷⁶⁸ - Ignore uninitialized chunks. Check proper indices.
- PR #3680¹⁷⁶⁹ - Ignore uninitialized chunks. Check proper range indices
- PR #3679¹⁷⁷⁰ - Simplify basic action implementations
- PR #3678¹⁷⁷¹ - Making sure `HPX_HAVE_LIBATOMIC` is unset before checking

¹⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3708>

¹⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3707>

¹⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3705>

¹⁷⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3704>

¹⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3703>

¹⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3702>

¹⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3700>

¹⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3697>

¹⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3694>

¹⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3693>

¹⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3692>

¹⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3691>

¹⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3690>

¹⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3688>

¹⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3687>

¹⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3686>

¹⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3685>

¹⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3684>

¹⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3683>

¹⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3682>

¹⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3680>

¹⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3679>

¹⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3678>

- PR #3677¹⁷⁷² - Fix generated full version number to be usable in expressions
- PR #3674¹⁷⁷³ - Reduce functional utilities call depth
- PR #3672¹⁷⁷⁴ - Change new build system to use existing macros related to pseudo dependencies
- PR #3669¹⁷⁷⁵ - Remove indirection in `function_ref` when thread description is disabled
- PR #3668¹⁷⁷⁶ - Unbreaking `async_*cb*` tests
- PR #3667¹⁷⁷⁷ - Generate `version.hpp`
- PR #3665¹⁷⁷⁸ - Enabling MPI parcelport for gitlab runners
- PR #3664¹⁷⁷⁹ - making clang-tidy work properly again
- PR #3662¹⁷⁸⁰ - Attempt to fix exception handling
- PR #3661¹⁷⁸¹ - Move `lcos::latch` to source file
- PR #3660¹⁷⁸² - Fix accidentally explicit `gid_type` default constructor
- PR #3659¹⁷⁸³ - Parallel executor latch
- PR #3658¹⁷⁸⁴ - Fixing `execution_parameters`
- PR #3657¹⁷⁸⁵ - Avoid dangling references in `wait_all`
- PR #3656¹⁷⁸⁶ - Avoiding lifetime problems with `sync_put_parcel`
- PR #3655¹⁷⁸⁷ - Fixing `nullptr` dereference inside of function
- PR #3652¹⁷⁸⁸ - Attempt to fix `thread_map_type` definition with C++11
- PR #3650¹⁷⁸⁹ - Allowing for end iterator being different from begin iterator
- PR #3649¹⁷⁹⁰ - Added architecture identification to `cmake` to be able to detect timestamp support
- PR #3645¹⁷⁹¹ - Enabling sanitizers on gitlab runner
- PR #3644¹⁷⁹² - Attempt to tackle timeouts during startup
- PR #3642¹⁷⁹³ - Cleanup parallel partitioners
- PR #3640¹⁷⁹⁴ - Dataflow now works with functions that return a reference

¹⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3677>

¹⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3674>

¹⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3672>

¹⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3669>

¹⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3668>

¹⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3667>

¹⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3665>

¹⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3664>

¹⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3662>

¹⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3661>

¹⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3660>

¹⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3659>

¹⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3658>

¹⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3657>

¹⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3656>

¹⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3655>

¹⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3652>

¹⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3650>

¹⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3649>

¹⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3645>

¹⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3644>

¹⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3642>

¹⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3640>

- PR #3637¹⁷⁹⁵ - Merging the executor-enabled overloads of `shared_future<>::then`
- PR #3633¹⁷⁹⁶ - Replace deprecated boost endian macros
- PR #3632¹⁷⁹⁷ - Add instructions on getting HPX to documentation
- PR #3631¹⁷⁹⁸ - Simplify parcel creation
- PR #3630¹⁷⁹⁹ - Small additions and fixes to release procedure
- PR #3629¹⁸⁰⁰ - Modular pp
- PR #3627¹⁸⁰¹ - Implement `util::function_ref`
- PR #3626¹⁸⁰² - Fix cancelable_action_client example
- PR #3625¹⁸⁰³ - Added automatic serialization for simple structs (see #3034)
- PR #3624¹⁸⁰⁴ - Updating the default order of priority for `thread_description`
- PR #3621¹⁸⁰⁵ - Update copyright year and other small formatting fixes
- PR #3620¹⁸⁰⁶ - Adding support for gitlab runner
- PR #3619¹⁸⁰⁷ - Store debug logs and core dumps on CircleCI
- PR #3618¹⁸⁰⁸ - Various optimizations
- PR #3617¹⁸⁰⁹ - Fix link to the gpg key (#2)
- PR #3615¹⁸¹⁰ - Fix unused variable warnings with networking off
- PR #3614¹⁸¹¹ - Restructuring counter data in scheduler to reduce false sharing
- PR #3613¹⁸¹² - Adding support for gitlab runners
- PR #3610¹⁸¹³ - Don't wait for `stop_condition` in main thread
- PR #3608¹⁸¹⁴ - Add inline keyword to `invalid_thread_id` definition for nvcc
- PR #3607¹⁸¹⁵ - Adding configuration key that allows one to explicitly add a directory to the component search path
- PR #3606¹⁸¹⁶ - Add nvcc to exclude constexpress since it is not supported by nvcc
- PR #3605¹⁸¹⁷ - Add inline to definition of checkpoint stream operators to fix link error

¹⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3637>

¹⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3633>

¹⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3632>

¹⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3631>

¹⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3630>

¹⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3629>

¹⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3627>

¹⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3626>

¹⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3625>

¹⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3624>

¹⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3621>

¹⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3620>

¹⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3619>

¹⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3618>

¹⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3617>

¹⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3615>

¹⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3614>

¹⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/3613>

¹⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3610>

¹⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3608>

¹⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3607>

¹⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3606>

¹⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3605>

- PR #3604¹⁸¹⁸ - Use format for string formatting
- PR #3603¹⁸¹⁹ - Improve the error message for using to less MAX_CPU_COUNT
- PR #3602¹⁸²⁰ - Improve the error message for to small values of MAX_CPU_COUNT
- PR #3600¹⁸²¹ - Parallel executor aggregated
- PR #3599¹⁸²² - Making sure networking is disabled for default one-locality-runs
- PR #3596¹⁸²³ - Store thread exit functions in `forward_list` instead of `deque` to avoid allocations
- PR #3590¹⁸²⁴ - Fix typo/mistake in thread queue `cleanup_terminated`
- PR #3588¹⁸²⁵ - Fix formatting errors in `launching_and_configuring_hpx_applications.rst`
- PR #3586¹⁸²⁶ - Make bind propagate value category
- PR #3585¹⁸²⁷ - Extend Cmake for building hpx as distribution packages (refs #3575)
- PR #3584¹⁸²⁸ - Untangle function storage from object pointer
- PR #3582¹⁸²⁹ - Towards Modularized HPX
- PR #3580¹⁸³⁰ - Remove extra `||` in `merge.hpp`
- PR #3577¹⁸³¹ - Partially revert “Remove vtable empty flag”
- PR #3576¹⁸³² - Make sure empty startup/shutdown functions are not being used
- PR #3574¹⁸³³ - Make sure DATAPAR settings are conveyed to depending projects
- PR #3573¹⁸³⁴ - Make sure HPX is usable with latest released version of Vc (V1.4.1)
- PR #3572¹⁸³⁵ - Adding test ensuring ticket 3565 is fixed
- PR #3571¹⁸³⁶ - Make empty `[unique_]` function vtable non-dependent
- PR #3566¹⁸³⁷ - Fix compilation with dynamic bitset for CPU masks
- PR #3563¹⁸³⁸ - Drop `util::[unique_]` function target_type
- PR #3562¹⁸³⁹ - Removing the target suffixes
- PR #3561¹⁸⁴⁰ - Replace executor traits return type deduction (keep non-SFINAE)

¹⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3604>

¹⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3603>

¹⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3602>

¹⁸²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3600>

¹⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/3599>

¹⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/3596>

¹⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3590>

¹⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3588>

¹⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3586>

¹⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3585>

¹⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3584>

¹⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3582>

¹⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3580>

¹⁸³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3577>

¹⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/3576>

¹⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/3574>

¹⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3573>

¹⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3572>

¹⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3571>

¹⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3566>

¹⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3563>

¹⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3562>

¹⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3561>

- PR #3557¹⁸⁴¹ - Replace the last usages of boost::atomic
- PR #3556¹⁸⁴² - Replace boost::scoped_array with std::unique_ptr
- PR #3552¹⁸⁴³ - (Re)move APEX readme
- PR #3548¹⁸⁴⁴ - Replace boost::scoped_ptr with std::unique_ptr
- PR #3547¹⁸⁴⁵ - Remove last use of Boost.Signals2
- PR #3544¹⁸⁴⁶ - Post 1.2.0 version bumps
- PR #3543¹⁸⁴⁷ - added Ubuntu dependency list to readme
- PR #3531¹⁸⁴⁸ - Warnings, warnings...
- PR #3527¹⁸⁴⁹ - Add CircleCI filter for building all tags
- PR #3525¹⁸⁵⁰ - Segmented algorithms
- PR #3517¹⁸⁵¹ - Replace boost::regex with C++11 <regex>
- PR #3514¹⁸⁵² - Cleaning up the build system
- PR #3505¹⁸⁵³ - Fixing type attribute warning for transfer_action
- PR #3504¹⁸⁵⁴ - Add support for rpm packaging
- PR #3499¹⁸⁵⁵ - Improving spinlock pools
- PR #3498¹⁸⁵⁶ - Remove thread specific ptr
- PR #3486¹⁸⁵⁷ - Fix comparison for expect_connecting_localities config entry
- PR #3469¹⁸⁵⁸ - Enable (existing) code for extracting stack pointer on Power platform

2.10.9 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.

¹⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3557>

¹⁸⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3556>

¹⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3552>

¹⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3548>

¹⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3547>

¹⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3544>

¹⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3543>

¹⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3531>

¹⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3527>

¹⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3525>

¹⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3517>

¹⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3514>

¹⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3505>

¹⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3504>

¹⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3499>

¹⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3498>

¹⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3486>

¹⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3469>

- Add support for Boost 1.69.0.

Closed issues

- Issue #3638¹⁸⁵⁹ - Build HPX 1.2 with boost 1.69
- Issue #3635¹⁸⁶⁰ - Non-deterministic crashing on Stampede2
- Issue #3550¹⁸⁶¹ - 1>e:000workhpxsrcthrow_exception.cpp(54): error C2440: ‘<function-style-cast>’: cannot convert from ‘boost::system::error_code’ to ‘hpx::exception’
- Issue #3549¹⁸⁶² - HPX 1.2.0 does not build on i686, but release candidate did
- Issue #3511¹⁸⁶³ - Build on s390x fails
- Issue #3509¹⁸⁶⁴ - Build on armv7l fails

Closed pull requests

- PR #3695¹⁸⁶⁵ - Don’t install CMake templates and packaging files
- PR #3666¹⁸⁶⁶ - Fixing yet another race in future_data
- PR #3663¹⁸⁶⁷ - Fixing race between setting and getting the value inside future_data
- PR #3648¹⁸⁶⁸ - Adding timestamp option for S390x platform
- PR #3647¹⁸⁶⁹ - Blind attempt to fix warnings issued by gcc V9
- PR #3611¹⁸⁷⁰ - Include GNUInstallDirs earlier to have it available for subdirectories
- PR #3595¹⁸⁷¹ - Use GNUInstallDirs lib path in pkgconfig config file
- PR #3593¹⁸⁷² - Add include(GNUInstallDirs) to HPXMacros.cmake
- PR #3591¹⁸⁷³ - Fix compilation error on arm7 architecture. Compiles and runs on Fedora 29 on Pi 3.
- PR #3558¹⁸⁷⁴ - Adding constructor *exception(boost::system::error_code const&)*
- PR #3555¹⁸⁷⁵ - cmake: make install locations configurable
- PR #3551¹⁸⁷⁶ - Fix uint64_t causing compilation fail on i686

¹⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3638>

¹⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3635>

¹⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/3550>

¹⁸⁶² <https://github.com/STELLAR-GROUP/hpx/issues/3549>

¹⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/3511>

¹⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3509>

¹⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3695>

¹⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3666>

¹⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3663>

¹⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3648>

¹⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3647>

¹⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3611>

¹⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3595>

¹⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3593>

¹⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3591>

¹⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3558>

¹⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3555>

¹⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3551>

2.10.10 HPX V1.2.0 (Nov 12, 2018)

General changes

Here are some of the main highlights and changes for this release:

- Thanks to the work of our Google Summer of Code student, Nikunj Gupta, we now have a new implementation of `hpx_main.hpp` on supported platforms (Linux, BSD and MacOS). This is intended to be a less fragile drop-in replacement for the old implementation relying on preprocessor macros. The new implementation does not require changes if you are using the [CMake¹⁸⁷⁷](#) or `pkg-config`. The old behaviour can be restored by setting `HPX_WITH_DYNAMIC_HPX_MAIN=OFF` during [CMake¹⁸⁷⁸](#) configuration. The implementation on Windows is unchanged.
- We have added functionality to allow passing scheduling hints to our schedulers. These will allow us to create executors that for example target a specific NUMA domain or allow for *HPX* threads to be pinned to a particular worker thread.
- We have significantly improved the performance of our futures implementation by making the shared state atomic.
- We have replaced Boostbook by Sphinx for our documentation. This means the documentation is easier to navigate with built-in search and table of contents. We have also added a quick start section and restructured the documentation to be easier to follow for new users.
- We have added a new option to the `--hpx:threads` command line option. It is now possible to use `cores` to tell *HPX* to only use one worker thread per core, unlike the existing option `all` which uses one worker thread per processing unit (processing unit can be a hyperthread if hyperthreads are available). The default value of `--hpx:threads` has also been changed to `cores` as this leads to better performance in most cases.
- All command line options can now be passed alongside configuration options when initializing *HPX*. This means that some options that were previously only available on the command line can now be set as configuration options.
- *HPXMP* is a portable, scalable, and flexible application programming interface using the OpenMP specification that supports multi-platform shared memory multiprocessing programming in C and C++. *HPXMP* can be enabled within *HPX* by setting `DHPX_WITH_HPXMP=ON` during [CMake¹⁸⁷⁹](#) 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 [CMake¹⁸⁸⁰](#) 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 [CMake¹⁸⁸¹](#) 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.

¹⁸⁷⁷ <https://www.cmake.org>

¹⁸⁷⁸ <https://www.cmake.org>

¹⁸⁷⁹ <https://www.cmake.org>

¹⁸⁸⁰ <https://www.cmake.org>

¹⁸⁸¹ <https://www.cmake.org>

Breaking changes

- The implementation of `hpx_main.hpp` has changed. If you are using custom Makefiles you will need to make changes. Please see the documentation on *using Makefiles* for more details.
- The default value of `--hpx:threads` has changed from `all` to `cores`. The new option `cores` only starts one worker thread per core.
- We have dropped support for Boost 1.56 and 1.57. The minimal version of Boost we now test is 1.58.
- Our `boost::format`-based formatting implementation has been revised and replaced with a custom implementation. This changes the formatting syntax and requires changes if you are relying on `hpx::util::format` or `hpx::util::format_to`. The pull request for this change contains more information: [PR #3266](#)¹⁸⁸².
- 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 #3538¹⁸⁸³ - numa handling incorrect for hwloc 2
- Issue #3533¹⁸⁸⁴ - Cmake version 3.5.1 does not work (git ff26b35 2018-11-06)
- Issue #3526¹⁸⁸⁵ - Failed building hpx-1.2.0-rc1 on Ubuntu16.04 x86-64 Virtualbox VM
- Issue #3512¹⁸⁸⁶ - Build on aarch64 fails
- Issue #3475¹⁸⁸⁷ - HPX fails to link if the MPI parcelport is enabled
- Issue #3462¹⁸⁸⁸ - CMake configuration shows a minor and inconsequential failure to create a symlink
- Issue #3461¹⁸⁸⁹ - Compilation Problems with the most recent Clang
- Issue #3460¹⁸⁹⁰ - Deadlock when create_partitioner fails (assertion fails) in debug mode
- Issue #3455¹⁸⁹¹ - HPX build failing with HWLOC errors on POWER8 with hwloc 1.8
- Issue #3438¹⁸⁹² - HPX no longer builds on IBM POWER8
- Issue #3426¹⁸⁹³ - hpx build failed on MacOS
- Issue #3424¹⁸⁹⁴ - CircleCI builds broken for forked repositories
- Issue #3422¹⁸⁹⁵ - Benchmarks in tests.performance.local are not run nightly

¹⁸⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3266>

¹⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/3538>

¹⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3533>

¹⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3526>

¹⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3512>

¹⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3475>

¹⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3462>

¹⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3461>

¹⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3460>

¹⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3455>

¹⁸⁹² <https://github.com/STELLAR-GROUP/hpx/issues/3438>

¹⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/3426>

¹⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3424>

¹⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3422>

- Issue #3408¹⁸⁹⁶ - CMake Targets for HPX
- Issue #3399¹⁸⁹⁷ - processing unit out of bounds
- Issue #3395¹⁸⁹⁸ - Floating point bug in hpx/runtime/threads/policies/scheduler_base.hpp
- Issue #3378¹⁸⁹⁹ - compile error with lcos::communicator
- Issue #3376¹⁹⁰⁰ - Failed to build HPX with APEX using clang
- Issue #3366¹⁹⁰¹ - Adapted Safe_Object example fails for -hpx:threads > 1
- Issue #3360¹⁹⁰² - Segmentation fault when passing component id as parameter
- Issue #3358¹⁹⁰³ - HPX runtime hangs after multiple (~thousands) start-stop sequences
- Issue #3352¹⁹⁰⁴ - Support TCP provider in libfabric ParcelPort
- Issue #3342¹⁹⁰⁵ - undefined reference to __atomic_load_16
- Issue #3339¹⁹⁰⁶ - setting command line options/flags from init cfg is not obvious
- Issue #3325¹⁹⁰⁷ - AGAS migrates components prematurely
- Issue #3321¹⁹⁰⁸ - hpx bad_parameter handling is awful
- Issue #3318¹⁹⁰⁹ - Benchmarks fail to build with C++11
- Issue #3304¹⁹¹⁰ - hpx::threads::run_as_hpx_thread does not properly handle exceptions
- Issue #3300¹⁹¹¹ - Setting pu step or offset results in no threads in default pool
- Issue #3297¹⁹¹² - Crash with APEX when running Phylanx lra_csv with > 1 thread
- Issue #3296¹⁹¹³ - Building HPX with APEX configuration gives compiler warnings
- Issue #3290¹⁹¹⁴ - make tests failing at hello_world_component
- Issue #3285¹⁹¹⁵ - possible compilation error when “using namespace std;” is defined before including “hpx” headers files
- Issue #3280¹⁹¹⁶ - HPX fails on OSX
- Issue #3272¹⁹¹⁷ - CircleCI does not upload generated docker image any more
- Issue #3270¹⁹¹⁸ - Error when compiling CUDA examples

¹⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3408>

¹⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3399>

¹⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3395>

¹⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3378>

¹⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3376>

¹⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/3366>

¹⁹⁰² <https://github.com/STELLAR-GROUP/hpx/issues/3360>

¹⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/3358>

¹⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3352>

¹⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3342>

¹⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3339>

¹⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3325>

¹⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3321>

¹⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3318>

¹⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3304>

¹⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3300>

¹⁹¹² <https://github.com/STELLAR-GROUP/hpx/issues/3297>

¹⁹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/3296>

¹⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3290>

¹⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3285>

¹⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3280>

¹⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3272>

¹⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3270>

- Issue #3267¹⁹¹⁹ - tests.unit.host_.block_allocator fails occasionally
- Issue #3264¹⁹²⁰ - Possible move to Sphinx for documentation
- Issue #3263¹⁹²¹ - Documentation improvements
- Issue #3259¹⁹²² - set_parcel_write_handler test fails occasionally
- Issue #3258¹⁹²³ - Links to source code in documentation are broken
- Issue #3247¹⁹²⁴ - Rare tests.unit.host_.block_allocator test failure on 1.1.0-rc1
- Issue #3244¹⁹²⁵ - Slowing down and speeding up an interval_timer
- Issue #3215¹⁹²⁶ - Cannot build both tests and examples on MSVC with pseudo-dependencies enabled
- Issue #3195¹⁹²⁷ - Unnecessary customization point route causing performance penalty
- Issue #3088¹⁹²⁸ - A strange thing in parallel::sort.
- Issue #2650¹⁹²⁹ - libfabric support for passive endpoints
- Issue #1205¹⁹³⁰ - TSS is broken

Closed pull requests

- PR #3542¹⁹³¹ - Fix numa lookup from pu when using hwloc 2.x
- PR #3541¹⁹³² - Fixing the build system of the MPI parcelport
- PR #3540¹⁹³³ - Updating HPX people section
- PR #3539¹⁹³⁴ - Splitting test to avoid OOM on CircleCI
- PR #3537¹⁹³⁵ - Fix guided exec
- PR #3536¹⁹³⁶ - Updating grants which support the LSU team
- PR #3535¹⁹³⁷ - Fix hiding of docker credentials
- PR #3534¹⁹³⁸ - Fixing #3533
- PR #3532¹⁹³⁹ - fixing minor doc typo --hpx:print-counter-at arg
- PR #3530¹⁹⁴⁰ - Changing APEX default tag to v2.1.0

¹⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3267>

¹⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3264>

¹⁹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/3263>

¹⁹²² <https://github.com/STELLAR-GROUP/hpx/issues/3259>

¹⁹²³ <https://github.com/STELLAR-GROUP/hpx/issues/3258>

¹⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3247>

¹⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3244>

1926 <https://github.com/STELLAR-GROUP/hpx/issues/3215>1927 <https://github.com/STELLAR-GROUP/hpx/issues/3195>1928 <https://github.com/STELLAR-GROUP/hpx/issues/3088>1929 <https://github.com/STELLAR-GROUP/hpx/issues/2650>1930 <https://github.com/STELLAR-GROUP/hpx/issues/1205>1931 <https://github.com/STELLAR-GROUP/hpx/pull/3542>1932 <https://github.com/STELLAR-GROUP/hpx/pull/3541>1933 <https://github.com/STELLAR-GROUP/hpx/pull/3540>1934 <https://github.com/STELLAR-GROUP/hpx/pull/3539>1935 <https://github.com/STELLAR-GROUP/hpx/pull/3537>1936 <https://github.com/STELLAR-GROUP/hpx/pull/3536>1937 <https://github.com/STELLAR-GROUP/hpx/pull/3535>1938 <https://github.com/STELLAR-GROUP/hpx/pull/3534>1939 <https://github.com/STELLAR-GROUP/hpx/pull/3532>1940 <https://github.com/STELLAR-GROUP/hpx/pull/3530>

- PR #3529¹⁹⁴¹ - Remove leftover security options and documentation
- PR #3528¹⁹⁴² - Fix hwloc version check
- PR #3524¹⁹⁴³ - Do not build guided pool examples with older GCC compilers
- PR #3523¹⁹⁴⁴ - Fix logging regression
- PR #3522¹⁹⁴⁵ - Fix more warnings
- PR #3521¹⁹⁴⁶ - Fixing argument handling in induction and reduction clauses for parallel::for_loop
- PR #3520¹⁹⁴⁷ - Remove docs symlink and versioned docs folders
- PR #3519¹⁹⁴⁸ - hpxMP release
- PR #3518¹⁹⁴⁹ - Change all steps to use new docker image on CircleCI
- PR #3516¹⁹⁵⁰ - Drop usage of deprecated facilities removed in C++17
- PR #3515¹⁹⁵¹ - Remove remaining uses of Boost.TypeTraits
- PR #3513¹⁹⁵² - Fixing a CMake problem when trying to use libfabric
- PR #3508¹⁹⁵³ - Remove memory_block component
- PR #3507¹⁹⁵⁴ - Propagating the MPI compile definitions to all relevant targets
- PR #3503¹⁹⁵⁵ - Update documentation colors and logo
- PR #3502¹⁹⁵⁶ - Fix bogus `throws` bindings in scheduled_thread_pool_impl
- PR #3501¹⁹⁵⁷ - Split parallel::remove_if tests to avoid OOM on CircleCI
- PR #3500¹⁹⁵⁸ - Support NONAMEPREFIX in add_hpx_library()
- PR #3497¹⁹⁵⁹ - Note that cuda support requires cmake 3.9
- PR #3495¹⁹⁶⁰ - Fixing dataflow
- PR #3493¹⁹⁶¹ - Remove deprecated options for 1.2.0 part 2
- PR #3492¹⁹⁶² - Add CUDA_LINK_LIBRARIES_KEYWORD to allow PRIVATE keyword in linkage t...
- PR #3491¹⁹⁶³ - Changing Base docker image

¹⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3529>

¹⁹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3528>

¹⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3524>

¹⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3523>

¹⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3522>

¹⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3521>

¹⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3520>

¹⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3519>

¹⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3518>

¹⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3516>

¹⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3515>

¹⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3513>

¹⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3508>

¹⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3507>

¹⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3503>

¹⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3502>

¹⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3501>

¹⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3500>

¹⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3497>

¹⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3495>

¹⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3493>

¹⁹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3492>

¹⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3491>

- PR #3490¹⁹⁶⁴ - Don't create tasks immediately with hpx::apply
- PR #3489¹⁹⁶⁵ - Remove deprecated options for 1.2.0
- PR #3488¹⁹⁶⁶ - Revert "Use BUILD_INTERFACE generator expression to fix cmake flag exports"
- PR #3487¹⁹⁶⁷ - Revert "Fixing type attribute warning for transfer_action"
- PR #3485¹⁹⁶⁸ - Use BUILD_INTERFACE generator expression to fix cmake flag exports
- PR #3483¹⁹⁶⁹ - Fixing type attribute warning for transfer_action
- PR #3481¹⁹⁷⁰ - Remove unused variables
- PR #3480¹⁹⁷¹ - Towards a more lightweight transfer action
- PR #3479¹⁹⁷² - Fix FLAGS - Use correct version of target_compile_options
- PR #3478¹⁹⁷³ - Making sure the application's exit code is properly propagated back to the OS
- PR #3476¹⁹⁷⁴ - Don't print docker credentials as part of the environment.
- PR #3473¹⁹⁷⁵ - Fixing invalid cmake code if no jemalloc prefix was given
- PR #3472¹⁹⁷⁶ - Attempting to work around recent clang test compilation failures
- PR #3471¹⁹⁷⁷ - Enable jemalloc on windows
- PR #3470¹⁹⁷⁸ - Updates readme
- PR #3468¹⁹⁷⁹ - Avoid hang if there is an exception thrown during startup
- PR #3467¹⁹⁸⁰ - Add compiler specific fallthrough attributes if C++17 attribute is not available
- PR #3466¹⁹⁸¹ - - bugfix : fix compilation with llvm-7.0
- PR #3465¹⁹⁸² - This patch adds various optimizations extracted from the thread_local_allocator work
- PR #3464¹⁹⁸³ - Check for forked repos in CircleCI docker push step
- PR #3463¹⁹⁸⁴ - - cmake : create the parent directory before symlinking
- PR #3459¹⁹⁸⁵ - Remove unused/incomplete functionality from util/logging
- PR #3458¹⁹⁸⁶ - Fix a problem with scope of CMAKE_CXX_FLAGS and hpx_add_compile_flag

¹⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3490>

¹⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3489>

¹⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3488>

¹⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3487>

¹⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3485>

¹⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3483>

¹⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3481>

¹⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3480>

¹⁹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3479>

¹⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3478>

¹⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3476>

¹⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3473>

¹⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3472>

¹⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3471>

¹⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3470>

¹⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3468>

¹⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3467>

¹⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3466>

¹⁹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3465>

¹⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3464>

¹⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3463>

¹⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3459>

¹⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3458>

- PR #3457¹⁹⁸⁷ - Fixing more size_t -> int16_t (and similar) warnings
- PR #3456¹⁹⁸⁸ - Add #ifdefs to topology.cpp to support old hwloc versions again
- PR #3454¹⁹⁸⁹ - Fixing warnings related to silent conversion of size_t -> int16_t
- PR #3451¹⁹⁹⁰ - Add examples as unit tests
- PR #3450¹⁹⁹¹ - Constexpr-fying bind and other functional facilities
- PR #3446¹⁹⁹² - Fix some thread suspension timeouts
- PR #3445¹⁹⁹³ - Fix various warnings
- PR #3443¹⁹⁹⁴ - Only enable service pool config options if pools are enabled
- PR #3441¹⁹⁹⁵ - Fix missing closing brackets in documentation
- PR #3439¹⁹⁹⁶ - Use correct MPI CXX libraries for MPI parcelport
- PR #3436¹⁹⁹⁷ - Add projection function to find_*(and fix very bad bug)
- PR #3435¹⁹⁹⁸ - Fixing 1205
- PR #3434¹⁹⁹⁹ - Fix threads cores
- PR #3433²⁰⁰⁰ - Add Heise Online to release announcement list
- PR #3432²⁰⁰¹ - Don't track task dependencies for distributed runs
- PR #3431²⁰⁰² - Circle CI setting changes for hpxMP
- PR #3430²⁰⁰³ - Fix unused params warning
- PR #3429²⁰⁰⁴ - One thread per core
- PR #3428²⁰⁰⁵ - This suppresses a deprecation warning that is being issued by MSVC 19.15.26726
- PR #3427²⁰⁰⁶ - Fixes #3426
- PR #3425²⁰⁰⁷ - Use source cache and workspace between job steps on CircleCI
- PR #3421²⁰⁰⁸ - Add CDash timing output to future overhead test (for graphs)
- PR #3420²⁰⁰⁹ - Add guided_pool_executor

¹⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3457>

¹⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3456>

¹⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3454>

¹⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3451>

¹⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3450>

¹⁹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3446>

¹⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3445>

¹⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3443>

¹⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3441>

1996 <https://github.com/STELLAR-GROUP/hpx/pull/3439>1997 <https://github.com/STELLAR-GROUP/hpx/pull/3436>1998 <https://github.com/STELLAR-GROUP/hpx/pull/3435>1999 <https://github.com/STELLAR-GROUP/hpx/pull/3434>2000 <https://github.com/STELLAR-GROUP/hpx/pull/3433>2001 <https://github.com/STELLAR-GROUP/hpx/pull/3432>2002 <https://github.com/STELLAR-GROUP/hpx/pull/3431>2003 <https://github.com/STELLAR-GROUP/hpx/pull/3430>2004 <https://github.com/STELLAR-GROUP/hpx/pull/3429>2005 <https://github.com/STELLAR-GROUP/hpx/pull/3428>2006 <https://github.com/STELLAR-GROUP/hpx/pull/3427>2007 <https://github.com/STELLAR-GROUP/hpx/pull/3425>2008 <https://github.com/STELLAR-GROUP/hpx/pull/3421>2009 <https://github.com/STELLAR-GROUP/hpx/pull/3420>

- PR #3419²⁰¹⁰ - Fix typo in CircleCI config
- PR #3418²⁰¹¹ - Add sphinx documentation
- PR #3415²⁰¹² - Scheduler NUMA hint and shared priority scheduler
- PR #3414²⁰¹³ - Adding step to synchronize the APEX release
- PR #3413²⁰¹⁴ - Fixing multiple defines of APEX_HAVE_HPX
- PR #3412²⁰¹⁵ - Fixes linking with libhpx_wrap error with BSD and Windows based systems
- PR #3410²⁰¹⁶ - Fix typo in CMakeLists.txt
- PR #3409²⁰¹⁷ - Fix brackets and indentation in existing_performance_counters.qbk
- PR #3407²⁰¹⁸ - Fix unused param and extra ; warnings emitted by gcc 8.x
- PR #3406²⁰¹⁹ - Adding thread local allocator and use it for future shared states
- PR #3405²⁰²⁰ - Adding DHPX_HAVE_THREAD_LOCAL_STORAGE=ON to builds
- PR #3404²⁰²¹ - fixing multiple definition of main() in linux
- PR #3402²⁰²² - Allow debug option to be enabled only for Linux systems with dynamic main on
- PR #3401²⁰²³ - Fix cuda_future_helper.h when compiling with C++11
- PR #3400²⁰²⁴ - Fix floating point exception scheduler_base idle backoff
- PR #3398²⁰²⁵ - Atomic future state
- PR #3397²⁰²⁶ - Fixing code for older gcc versions
- PR #3396²⁰²⁷ - Allowing to register thread event functions (start/stop/error)
- PR #3394²⁰²⁸ - Fix small mistake in primary_namespace_server.cpp
- PR #3393²⁰²⁹ - Explicitly instantiate configured schedulers
- PR #3392²⁰³⁰ - Add performance counters background overhead and background work duration
- PR #3391²⁰³¹ - Adapt integration of HPXMP to latest build system changes
- PR #3390²⁰³² - Make AGAS measurements optional

²⁰¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3419>

²⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3418>

²⁰¹² <https://github.com/STELLAR-GROUP/hpx/pull/3415>

²⁰¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3414>

²⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3413>

²⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3412>

²⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3410>

²⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3409>

²⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3407>

²⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3406>

²⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3405>

²⁰²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3404>

²⁰²² <https://github.com/STELLAR-GROUP/hpx/pull/3402>

²⁰²³ <https://github.com/STELLAR-GROUP/hpx/pull/3401>

²⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3400>

²⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3398>

²⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3397>

²⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3396>

²⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3394>

²⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3393>

²⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3392>

²⁰³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3391>

²⁰³² <https://github.com/STELLAR-GROUP/hpx/pull/3390>

- PR #3389²⁰³³ - Fix deadlock during shutdown
- PR #3388²⁰³⁴ - Add several functionalities allowing to optimize synchronous action invocation
- PR #3387²⁰³⁵ - Add cmake option to opt out of fail-compile tests
- PR #3386²⁰³⁶ - Adding support for boost::container::small_vector to dataflow
- PR #3385²⁰³⁷ - Adds Debug option for hpx initializing from main
- PR #3384²⁰³⁸ - This hopefully fixes two tests that occasionally fail
- PR #3383²⁰³⁹ - Making sure thread local storage is enable for hpxMP
- PR #3382²⁰⁴⁰ - Fix usage of HPX_CAPTURE together with default value capture [=]
- PR #3381²⁰⁴¹ - Replace undefined instantiations of uniform_int_distribution
- PR #3380²⁰⁴² - Add missing semicolons to uses of HPX_COMPILER_FENCE
- PR #3379²⁰⁴³ - Fixing #3378
- PR #3377²⁰⁴⁴ - Adding build system support to integrate hpxmp into hpx at the user's machine
- PR #3375²⁰⁴⁵ - Replacing wrapper for __libc_start_main with main
- PR #3374²⁰⁴⁶ - Adds hpx_wrap to HPX_LINK_LIBRARIES which links only when specified.
- PR #3373²⁰⁴⁷ - Forcing cache settings in HPXConfig.cmake to guarantee updated values
- PR #3372²⁰⁴⁸ - Fix some more c++11 build problems
- PR #3371²⁰⁴⁹ - Adds HPX_LINKER_FLAGS to HPX applications without editing their source codes
- PR #3370²⁰⁵⁰ - util::format: add typeSpecifier<> specializations for %!s(MISSING) and %!!(MISSING)s
- PR #3369²⁰⁵¹ - Adding configuration option to allow explicit disable of the new hpx_main feature on Linux
- PR #3368²⁰⁵² - Updates doc with recent hpx_wrap implementation
- PR #3367²⁰⁵³ - Adds Mac OS implementation to hpx_main.hpp
- PR #3365²⁰⁵⁴ - Fix order of hpx libs in HPX_CONF_LIBRARIES.
- PR #3363²⁰⁵⁵ - Apex fixing null wrapper

²⁰³³ <https://github.com/STELLAR-GROUP/hpx/pull/3389>

²⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3388>

²⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3387>

²⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3386>

²⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3385>

²⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3384>

²⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3383>

²⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3382>

²⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3381>

²⁰⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3380>

²⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3379>

²⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3377>

²⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3375>

²⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3374>

²⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3373>

²⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3372>

²⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3371>

²⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3370>

²⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3369>

²⁰⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3368>

²⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3367>

²⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3365>

²⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3363>

- PR #3361²⁰⁵⁶ - Making sure all parcels get destroyed on an HPX thread (TCP pp)
- PR #3359²⁰⁵⁷ - Feature/improveerrorforcompiler
- PR #3357²⁰⁵⁸ - Static/dynamic executable implementation
- PR #3355²⁰⁵⁹ - Reverting changes introduced by #3283 as those make applications hang
- PR #3354²⁰⁶⁰ - Add external dependencies to HPX_LIBRARY_DIR
- PR #3353²⁰⁶¹ - Fix libfabric tcp
- PR #3351²⁰⁶² - Move obsolete header to tests directory.
- PR #3350²⁰⁶³ - Renaming two functions to avoid problem described in #3285
- PR #3349²⁰⁶⁴ - Make idle backoff exponential with maximum sleep time
- PR #3347²⁰⁶⁵ - Replace *simple_component** with *component** in the Documentation
- PR #3346²⁰⁶⁶ - Fix CMakeLists.txt example in quick start
- PR #3345²⁰⁶⁷ - Fix automatic setting of HPX_MORE_THAN_64_THREADS
- PR #3344²⁰⁶⁸ - Reduce amount of information printed for unknown command line options
- PR #3343²⁰⁶⁹ - Safeguard HPX against destruction in global contexts
- PR #3341²⁰⁷⁰ - Allowing for all command line options to be used as configuration settings
- PR #3340²⁰⁷¹ - Always convert inspect results to JUnit XML
- PR #3336²⁰⁷² - Only run docker push on master on CircleCI
- PR #3335²⁰⁷³ - Update description of hpx.os_threads config parameter.
- PR #3334²⁰⁷⁴ - Making sure early logging settings don't get mixed with others
- PR #3333²⁰⁷⁵ - Update CMake links and versions in documentation
- PR #3332²⁰⁷⁶ - Add notes on target suffixes to CMake documentation
- PR #3331²⁰⁷⁷ - Add quickstart section to documentation
- PR #3330²⁰⁷⁸ - Rename resource_partitioner test to avoid conflicts with pseudodependencies

²⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3361>

²⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3359>

²⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3357>

²⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3355>

²⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3354>

²⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3353>

²⁰⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3351>

²⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3350>

²⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3349>

²⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3347>

²⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3346>

²⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3345>

²⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3344>

²⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3343>

²⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3341>

²⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3340>

²⁰⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3336>

²⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3335>

²⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3334>

²⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3333>

²⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3332>

²⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3331>

²⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3330>

- PR #3328²⁰⁷⁹ - Making sure object is pinned while executing actions, even if action returns a future
- PR #3327²⁰⁸⁰ - Add missing std::forward to tuple.hpp
- PR #3326²⁰⁸¹ - Make sure logging is up and running while modules are being discovered.
- PR #3324²⁰⁸² - Replace C++14 overload of std::equal with C++11 code.
- PR #3323²⁰⁸³ - Fix a missing apex thread data (wrapper) initialization
- PR #3320²⁰⁸⁴ - Adding support for -std=c++2a (define *HPX_WITH_CXX2A=On*)
- PR #3319²⁰⁸⁵ - Replacing C++14 feature with equivalent C++11 code
- PR #3317²⁰⁸⁶ - Fix compilation with VS 15.7.1 and /std:c++latest
- PR #3316²⁰⁸⁷ - Fix includes for 1d_stencil_*_omp examples
- PR #3314²⁰⁸⁸ - Remove some unused parameter warnings
- PR #3313²⁰⁸⁹ - Fix pu-step and pu-offset command line options
- PR #3312²⁰⁹⁰ - Add conversion of inspect reports to JUnit XML
- PR #3311²⁰⁹¹ - Fix escaping of closing braces in format specification syntax
- PR #3310²⁰⁹² - Don't overwrite user settings with defaults in registration database
- PR #3309²⁰⁹³ - Fixing potential stack overflow for dataflow
- PR #3308²⁰⁹⁴ - This updates the .clang-format configuration file to utilize newer features
- PR #3306²⁰⁹⁵ - Marking migratable objects in their gid to allow not handling migration in AGAS
- PR #3305²⁰⁹⁶ - Add proper exception handling to run_as_hpx_thread
- PR #3303²⁰⁹⁷ - Changed std::rand to a better inbuilt PRNG Generator
- PR #3302²⁰⁹⁸ - All non-migratable (simple) components now encode their lva and component type in their gid
- PR #3301²⁰⁹⁹ - Add nullptr_t overloads to resource partitioner
- PR #3298²¹⁰⁰ - Apex task wrapper memory bug
- PR #3295²¹⁰¹ - Fix mistakes after merge of CircleCI config

²⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3328>

²⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3327>

²⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3326>

²⁰⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3324>

²⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3323>

²⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3320>

²⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3319>

²⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3317>

²⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3316>

²⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3314>

²⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3313>

²⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3312>

²⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3311>

²⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3310>

²⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3309>

²⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3308>

²⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3306>

²⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3305>

²⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3303>

²⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3302>

²⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3301>

²¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3298>

²¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3295>

- PR #3294²¹⁰² - Fix partitioned vector include in partitioned_vector_find tests
- PR #3293²¹⁰³ - Adding emplace support to promise and make_ready_future
- PR #3292²¹⁰⁴ - Add new cuda kernel synchronization with hpx::future demo
- PR #3291²¹⁰⁵ - Fixes #3290
- PR #3289²¹⁰⁶ - Fixing Docker image creation
- PR #3288²¹⁰⁷ - Avoid allocating shared state for wait_all
- PR #3287²¹⁰⁸ - Fixing /scheduler/utilization/instantaneous performance counter
- PR #3286²¹⁰⁹ - dataflow() and future::then() use sync policy where possible
- PR #3284²¹¹⁰ - Background thread can use relaxed atomics to manipulate thread state
- PR #3283²¹¹¹ - Do not unwrap ready future
- PR #3282²¹¹² - Fix virtual method override warnings in static schedulers
- PR #3281²¹¹³ - Disable set_area_membind_nodeset for OSX
- PR #3279²¹¹⁴ - Add two variations to the future_overhead benchmark
- PR #3278²¹¹⁵ - Fix circleci workspace
- PR #3277²¹¹⁶ - Support external plugins
- PR #3276²¹¹⁷ - Fix missing parenthesis in hello_compute.cu.
- PR #3274²¹¹⁸ - Reinit counters synchronously in reinit_counters test
- PR #3273²¹¹⁹ - Splitting tests to avoid compiler OOM
- PR #3271²¹²⁰ - Remove leftover code from context_generic_context.hpp
- PR #3269²¹²¹ - Fix bulk_construct with count = 0
- PR #3268²¹²² - Replace constexpr with HPX_CXX14_CONSTEXPR and HPX_CONSTEXPR
- PR #3266²¹²³ - Replace boost::format with custom sprintf-based implementation
- PR #3265²¹²⁴ - Split parallel tests on CircleCI

²¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3294>

²¹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3293>

²¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3292>

²¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3291>

²¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3289>

²¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3288>

²¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3287>

²¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3286>

²¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3284>

²¹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3283>

²¹¹² <https://github.com/STELLAR-GROUP/hpx/pull/3282>

²¹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3281>

²¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3279>

²¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3278>

²¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3277>

²¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3276>

²¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3274>

²¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3273>

²¹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3271>

²¹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3269>

²¹²² <https://github.com/STELLAR-GROUP/hpx/pull/3268>

²¹²³ <https://github.com/STELLAR-GROUP/hpx/pull/3266>

²¹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3265>

- PR #3262²¹²⁵ - Making sure documentation correctly links to source files
- PR #3261²¹²⁶ - Apex refactoring fix rebind
- PR #3260²¹²⁷ - Isolate performance counter parser into a separate TU
- PR #3256²¹²⁸ - Post 1.1.0 version bumps
- PR #3254²¹²⁹ - Adding trait for actions allowing to make runtime decision on whether to execute it directly
- PR #3253²¹³⁰ - Bump minimal supported Boost to 1.58.0
- PR #3251²¹³¹ - Adds new feature: changing interval used in interval_timer (issue 3244)
- PR #3239²¹³² - Changing std::rand() to a better inbuilt PRNG generator.
- PR #3234²¹³³ - Disable background thread when networking is off
- PR #3232²¹³⁴ - Clean up suspension tests
- PR #3230²¹³⁵ - Add optional scheduler mode parameter to create_thread_pool function
- PR #3228²¹³⁶ - Allow suspension also on static schedulers
- PR #3163²¹³⁷ - libfabric parcelport w/o HPX_PARCELPORT_LIBFABRIC_ENDPOINT_RDM
- PR #3036²¹³⁸ - Switching to CircleCI 2.0

2.10.11 HPX V1.1.0 (Mar 24, 2018)

General changes

Here are some of the main highlights and changes for this release (in no particular order):

- We have changed the way *HPX* manages the processing units on a node. We do not longer implicitly bind all available cores to a single thread pool. The user has now full control over what processing units are bound to what thread pool, each with a separate scheduler. It is now also possible to create your own scheduler implementation and control what processing units this scheduler should use. We added the `hpx::resource::partitioner` that manages all available processing units and assigns resources to the used thread pools. Thread pools can be now be suspended/resumed independently. This functionality helps in running *HPX* concurrently to code that is directly relying on OpenMP²¹³⁹ and/or MPI²¹⁴⁰.
- We have continued to implement various parallel algorithms. *HPX* now almost completely implements all of the parallel algorithms as specified by the C++17 Standard²¹⁴¹. We have also continued to implement these algorithms for the distributed use case (for segmented data structures, such as `hpx::partitioned_vector`).

²¹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3262>

²¹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3261>

²¹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3260>

²¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3256>

²¹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3254>

²¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3253>

²¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3251>

²¹³² <https://github.com/STELLAR-GROUP/hpx/pull/3239>

²¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/3234>

²¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3232>

²¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3230>

²¹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3228>

²¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3163>

²¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3036>

²¹³⁹ <https://openmp.org/wp/>

²¹⁴⁰ https://en.wikipedia.org/wiki/Message_Passing_Interface

²¹⁴¹ <http://www.open-std.org/jtc1/sc22/wg21>

- Added a compatibility layer for `std::thread`, `std::mutex`, and `std::condition_variable` allowing for the code to use those facilities where available and to fall back to the corresponding Boost facilities otherwise. The CMake²¹⁴² 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 CMake²¹⁴³.
- The parameter sequence for the `hpx::parallel::inclusive_scan` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17. The old overloads can be still enabled at configure time by passing `-DHPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY=On` to CMake.
- Added a helper facility `hpx::local_new` which is equivalent to `hpx::new_` except that it creates components locally only. As a consequence, the used component constructor may accept non-serializable argument types and/or non-const references or pointers.
- Removed the (broken) component type `hpx::lcos::queue<T>`. The old type is still available at configure time by passing `-DHPX_WITH_QUEUE_COMPATIBILITY=On` to CMake.
- The parallel algorithms adopted for C++17 restrict the iterator categories usable with those to at least forward iterators. Our implementation of the parallel algorithms was supporting input iterators (and output iterators) as well by simply falling back to sequential execution. We have now made our implementations conforming by requiring at least forward iterators. In order to enable the old behavior use the compatibility option `-DHPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT=On` on the CMake²¹⁴⁴ command line.
- We have added the functionalities allowing for LCOs being implemented using (simple) components. Before LCOs had to always be implemented using managed components.
- User defined components don't have to be default-constructible anymore. Return types from actions don't have to be default-constructible anymore either. Our serialization layer now in general supports non-default-constructible types.
- We have added a new launch policy `hpx::launch::lazy` that allows one to defer the decision on what launch policy to use to the point of execution. This policy is initialized with a function (object) that – when invoked – is expected to produce the desired launch policy.

Breaking changes

- We have dropped support for the gcc compiler version V4.8. The minimal gcc version we now test on is gcc V4.9. The minimally required version of CMake²¹⁴⁵ 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`.

²¹⁴² <https://www.cmake.org>

²¹⁴³ <https://www.cmake.org>

²¹⁴⁴ <https://www.cmake.org>

²¹⁴⁵ <https://www.cmake.org>

- 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 #3250²¹⁴⁶ - Apex refactoring with guids
- PR #3249²¹⁴⁷ - Updating People.qbk
- PR #3246²¹⁴⁸ - Assorted fixes for CUDA
- PR #3245²¹⁴⁹ - Apex refactoring with guids
- PR #3242²¹⁵⁰ - Modify task counting in `thread_queue.hpp`
- PR #3240²¹⁵¹ - Fixed typos
- PR #3238²¹⁵² - Readding accidentally removed `std::abort`
- PR #3237²¹⁵³ - Adding Pipeline example
- PR #3236²¹⁵⁴ - Fixing `memory_block`
- PR #3233²¹⁵⁵ - Make `schedule_thread` take suspended threads into account
- Issue #3226²¹⁵⁶ - `memory_block` is breaking, signaling SIGSEGV on a thread on creation and freeing
- PR #3225²¹⁵⁷ - Applying quick fix for hwloc-2.0
- Issue #3224²¹⁵⁸ - HPX counters crashing the application
- PR #3223²¹⁵⁹ - Fix returns when setting config entries
- Issue #3222²¹⁶⁰ - Errors linking `libhpx.so`

²¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3250>

²¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3249>

²¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3246>

²¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3245>

²¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3242>

²¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3240>

²¹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3238>

²¹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3237>

²¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3236>

²¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3233>

²¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3226>

2157 <https://github.com/STELLAR-GROUP/hpx/pull/3225>2158 <https://github.com/STELLAR-GROUP/hpx/issues/3224>2159 <https://github.com/STELLAR-GROUP/hpx/pull/3223>2160 <https://github.com/STELLAR-GROUP/hpx/issues/3222>

- Issue #3221²¹⁶¹ - HPX on Mac OS X with HWLoc 2.0.0 fails to run
- PR #3216²¹⁶² - Reorder a variadic array to satisfy VS 2017 15.6
- PR #3214²¹⁶³ - Changed prerequisites.qbk to avoid confusion while building boost
- PR #3213²¹⁶⁴ - Relax locks for thread suspension to avoid holding locks when yielding
- PR #3212²¹⁶⁵ - Fix check in sequenced_executor test
- PR #3211²¹⁶⁶ - Use preinit_array to set argc/argv in init_globally example
- PR #3210²¹⁶⁷ - Adapted parallel::{search | search_n} for Ranges TS (see #1668)
- PR #3209²¹⁶⁸ - Fix locking problems during shutdown
- Issue #3208²¹⁶⁹ - init_globally throwing a run-time error
- PR #3206²¹⁷⁰ - Addition of new arithmetic performance counter “Count”
- PR #3205²¹⁷¹ - Fixing return type calculation for bulk_then_execute
- PR #3204²¹⁷² - Changing std::rand() to a better inbuilt PRNG generator
- PR #3203²¹⁷³ - Resolving problems during shutdown for VS2015
- PR #3202²¹⁷⁴ - Making sure resource partitioner is not accessed if its not valid
- PR #3201²¹⁷⁵ - Fixing optional::swap
- Issue #3200²¹⁷⁶ - hpx::util::optional fails
- PR #3199²¹⁷⁷ - Fix sliding_semaphore test
- PR #3198²¹⁷⁸ - Set pre_main status before launching run_helper
- PR #3197²¹⁷⁹ - Update README.rst
- PR #3194²¹⁸⁰ - parallel::{fill|fill_n} updated for Ranges TS
- PR #3193²¹⁸¹ - Updating Runtime.cpp by adding correct description of Performance counters during register
- PR #3191²¹⁸² - Fix sliding_semaphore_2338 test
- PR #3190²¹⁸³ - Topology improvements

²¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/3221>

²¹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3216>

²¹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3214>

²¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3213>

²¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3212>

²¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3211>

²¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3210>

²¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3209>

²¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3208>

²¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3206>

²¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3205>

²¹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3204>

²¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3203>

²¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3202>

²¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3201>

²¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3200>

²¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3199>

2178 <https://github.com/STELLAR-GROUP/hpx/pull/3198>2179 <https://github.com/STELLAR-GROUP/hpx/pull/3197>2180 <https://github.com/STELLAR-GROUP/hpx/pull/3194>2181 <https://github.com/STELLAR-GROUP/hpx/pull/3193>2182 <https://github.com/STELLAR-GROUP/hpx/pull/3191>2183 <https://github.com/STELLAR-GROUP/hpx/pull/3190>

- PR #3189²¹⁸⁴ - Deleting one include of median from BOOST library to arithmetics_counter file
- PR #3188²¹⁸⁵ - Optionally disable printing of diagnostics during terminate
- PR #3187²¹⁸⁶ - Suppressing cmake warning issued by cmake > V3.11
- PR #3185²¹⁸⁷ - Remove unused scoped_unlock, unlock_guard_try
- PR #3184²¹⁸⁸ - Fix nqueen example
- PR #3183²¹⁸⁹ - Add runtime start/stop, resume/suspend and OpenMP benchmarks
- Issue #3182²¹⁹⁰ - bulk_then_execute has unexpected return type/does not compile
- Issue #3181²¹⁹¹ - hwloc 2.0 breaks topo class and cannot be used
- Issue #3180²¹⁹² - Schedulers that don't support suspend/resume are unusable
- PR #3179²¹⁹³ - Various minor changes to support FLeCSI
- PR #3178²¹⁹⁴ - Fix #3124
- PR #3177²¹⁹⁵ - Removed allgather
- PR #3176²¹⁹⁶ - Fixed Documentation for "using_hpx_pkgconfig"
- PR #3174²¹⁹⁷ - Add hpx::iostreams::ostream overload to format_to
- PR #3172²¹⁹⁸ - Fix lifo queue backend
- PR #3171²¹⁹⁹ - adding the missing unset() function to cpu_mask() for case of more than 64 threads
- PR #3170²²⁰⁰ - Add cmake flag -DHPX_WITHFAULT_TOLERANCE=ON (OFF by default)
- PR #3169²²⁰¹ - Adapted parallel::{countlcount_if} for Ranges TS (see #1668)
- PR #3168²²⁰² - Changing used namespace for seq execution policy
- Issue #3167²²⁰³ - Update GSoC projects
- Issue #3166²²⁰⁴ - Application (Octotiger) gets stuck on hpx::finalize when only using one thread
- Issue #3165²²⁰⁵ - Compilation of parallel algorithms with HPX_WITH_DATAPAR is broken
- PR #3164²²⁰⁶ - Fixing component migration

²¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3189>

²¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3188>

²¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3187>

²¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3185>

²¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3184>

²¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3183>

²¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3182>

²¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3181>

²¹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/3180>

²¹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3179>

²¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3178>

²¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3177>

²¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3176>

²¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3174>

²¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3172>

²¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3171>

²²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3170>

²²⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3169>

²²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3168>

²²⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/3167>

²²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3166>

²²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3165>

²²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3164>

- PR #3162²²⁰⁷ - regex_from_pattern: escape regex special characters to avoid misinterpretation
- Issue #3161²²⁰⁸ - Building HPX with hwloc 2.0.0 fails
- PR #3160²²⁰⁹ - Fixing the handling of quoted command line arguments.
- PR #3158²²¹⁰ - Fixing a race with timed suspension (second attempt)
- PR #3157²²¹¹ - Revert “Fixing a race with timed suspension”
- PR #3156²²¹² - Fixing serialization of classes with incompatible serialize signature
- PR #3154²²¹³ - More refactorings based on clang-tidy reports
- PR #3153²²¹⁴ - Fixing a race with timed suspension
- PR #3152²²¹⁵ - Documentation for runtime suspension
- PR #3151²²¹⁶ - Use small_vector only from boost version 1.59 onwards
- PR #3150²²¹⁷ - Avoiding more stack overflows
- PR #3148²²¹⁸ - Refactoring component_base and base_action/transfer_base_action
- PR #3147²²¹⁹ - Move yield_while out of detail namespace and into own file
- PR #3145²²²⁰ - Remove a leftover of the cxx11 std array cleanup
- PR #3144²²²¹ - Minor changes to how actions are executed
- PR #3143²²²² - Fix stack overhead
- PR #3142²²²³ - Fix typo in config.hpp
- PR #3141²²²⁴ - Fixing small_vector compatibility with older boost version
- PR #3140²²²⁵ - is_heap_text fix
- Issue #3139²²²⁶ - Error in is_heap_tests.hpp
- PR #3138²²²⁷ - Partially reverting #3126
- PR #3137²²²⁸ - Suspend speedup
- PR #3136²²²⁹ - Revert “Fixing #2325”

²²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3162>

²²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3161>

²²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3160>

²²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3158>

²²¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3157>

²²¹² <https://github.com/STELLAR-GROUP/hpx/pull/3156>

²²¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3154>

²²¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3153>

²²¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3152>

²²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3151>

²²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3150>

²²¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3148>

²²¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3147>

²²²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3145>

²²²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3144>

²²²² <https://github.com/STELLAR-GROUP/hpx/pull/3143>

²²²³ <https://github.com/STELLAR-GROUP/hpx/pull/3142>

²²²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3141>

²²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3140>

²²²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3139>

²²²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3138>

²²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3137>

²²²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3136>

- PR #3135²²³⁰ - Improving destruction of threads
- Issue #3134²²³¹ - HPX_SERIALIZATION_SPLIT_FREE does not stop compiler from looking for serialize() method
- PR #3133²²³² - Make hwloc compulsory
- PR #3132²²³³ - Update CXX14 constexpr feature test
- PR #3131²²³⁴ - Fixing #2325
- PR #3130²²³⁵ - Avoid completion handler allocation
- PR #3129²²³⁶ - Suspend runtime
- PR #3128²²³⁷ - Make docbook dtd and xsl path names consistent
- PR #3127²²³⁸ - Add hpx::start nullptr overloads
- PR #3126²²³⁹ - Cleaning up coroutine implementation
- PR #3125²²⁴⁰ - Replacing nullptr with hpx::threads::invalid_thread_id
- Issue #3124²²⁴¹ - Add hello_world_component to CI builds
- PR #3123²²⁴² - Add new constructor.
- PR #3122²²⁴³ - Fixing #3121
- Issue #3121²²⁴⁴ - HPX_SMT_PAUSE is broken on non-x86 platforms when __GNUC__ is defined
- PR #3120²²⁴⁵ - Don't use boost::intrusive_ptr for thread_id_type
- PR #3119²²⁴⁶ - Disable default executor compatibility with V1 executors
- PR #3118²²⁴⁷ - Adding performance_counter::reinit to allow for dynamically changing counter sets
- PR #3117²²⁴⁸ - Replace uses of boost/experimental::optional with util::optional
- PR #3116²²⁴⁹ - Moving background thread APEX timer #2980
- PR #3115²²⁵⁰ - Fixing race condition in channel test
- PR #3114²²⁵¹ - Avoid using util::function for thread function wrappers
- PR #3113²²⁵² - cmake V3.10.2 has changed the variable names used for MPI

²²³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3135>

²²³¹ <https://github.com/STELLAR-GROUP/hpx/issues/3134>

²²³² <https://github.com/STELLAR-GROUP/hpx/pull/3133>

²²³³ <https://github.com/STELLAR-GROUP/hpx/pull/3132>

²²³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3131>

²²³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3130>

²²³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3129>

²²³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3128>

²²³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3127>

²²³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3126>

²²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3125>

²²⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/3124>

²²⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3123>

²²⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3122>

²²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3121>

²²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3120>

²²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3119>

²²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3118>

²²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3117>

²²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3116>

²²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3115>

²²⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3114>

²²⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3113>

- PR #3112²²⁵³ - Minor fixes to exclusive_scan algorithm
- PR #3111²²⁵⁴ - Revert “fix detection of cxx11_std_atomic”
- PR #3110²²⁵⁵ - Suspend thread pool
- PR #3109²²⁵⁶ - Fixing thread scheduling when yielding a thread id
- PR #3108²²⁵⁷ - Revert “Suspend thread pool”
- PR #3107²²⁵⁸ - Remove UB from thread::id relational operators
- PR #3106²²⁵⁹ - Add cmake test for std::decay_t to fix cuda build
- PR #3105²²⁶⁰ - Fixing refcount for async traversal frame
- PR #3104²²⁶¹ - Local execution of direct actions is now actually performed directly
- PR #3103²²⁶² - Adding support for generic counter_raw_values performance counter type
- Issue #3102²²⁶³ - Introduce generic performance counter type returning an array of values
- PR #3101²²⁶⁴ - Revert “Adapting stack overhead limit for gcc 4.9”
- PR #3100²²⁶⁵ - Fix #3068 (condition_variable deadlock)
- PR #3099²²⁶⁶ - Fixing lock held during suspension in papi counter component
- PR #3098²²⁶⁷ - Unbreak broadcast_wait_for_2822 test
- PR #3097²²⁶⁸ - Adapting stack overhead limit for gcc 4.9
- PR #3096²²⁶⁹ - fix detection of cxx11_std_atomic
- PR #3095²²⁷⁰ - Add ciso646 header to get _LIBCPP_VERSION for testing inplace merge
- PR #3094²²⁷¹ - Relax atomic operations on performance counter values
- PR #3093²²⁷² - Short-circuit all_of/any_of/none_of instantiations
- PR #3092²²⁷³ - Take advantage of C++14 lambda capture initialization syntax, where possible
- PR #3091²²⁷⁴ - Remove more references to Boost from logging code
- PR #3090²²⁷⁵ - Unify use of yield/yield_k

²²⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3112>

²²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3111>

²²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3110>

²²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3109>

²²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3108>

²²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3107>

²²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3106>

²²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3105>

²²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3104>

²²⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3103>

²²⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/3102>

²²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3101>

²²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3100>

²²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3099>

²²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3098>

²²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3097>

²²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3096>

²²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3095>

²²⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3094>

²²⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3093>

²²⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3092>

²²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3091>

²²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3090>

- PR #3089²²⁷⁶ - Fix a strange thing in parallel::detail::handle_exception. (Fix #2834.)
- Issue #3088²²⁷⁷ - A strange thing in parallel::sort.
- PR #3087²²⁷⁸ - Fixing assertion in default_distribution_policy
- PR #3086²²⁷⁹ - Implement parallel::remove and parallel::remove_if
- PR #3085²²⁸⁰ - Addressing breaking changes in Boost V1.66
- PR #3084²²⁸¹ - Ignore build warnings round 2
- PR #3083²²⁸² - Fix typo HPX_WITH_MM_PREFECHT
- PR #3081²²⁸³ - Pre-decay template arguments early
- PR #3080²²⁸⁴ - Suspend thread pool
- PR #3079²²⁸⁵ - Ignore build warnings
- PR #3078²²⁸⁶ - Don't test inplace_merge with libc++
- PR #3076²²⁸⁷ - Fixing 3075: Part 1
- PR #3074²²⁸⁸ - Fix more build warnings
- PR #3073²²⁸⁹ - Suspend thread cleanup
- PR #3072²²⁹⁰ - Change existing symbol_namespace::iterate to return all data instead of invoking a callback
- PR #3071²²⁹¹ - Fixing pack_traversal_async test
- PR #3070²²⁹² - Fix dynamic_counters_loaded_1508 test by adding dependency to memory_component
- PR #3069²²⁹³ - Fix scheduling loop exit
- Issue #3068²²⁹⁴ - hpx::lcos::condition_variable could be suspect to deadlocks
- PR #3067²²⁹⁵ - #ifdef out random_shuffle deprecated in later c++
- PR #3066²²⁹⁶ - Make coalescing test depend on coalescing library to ensure it gets built
- PR #3065²²⁹⁷ - Workaround for minimal_timed_async_executor_test compilation failures, attempts to copy a deferred call (in unevaluated context)
- PR #3064²²⁹⁸ - Fixing wrong condition in wrapper_heap

²²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3089>

²²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3088>

²²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3087>

²²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3086>

²²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3085>

²²⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3084>

²²⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3083>

²²⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3081>

²²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3080>

²²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3079>

²²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3078>

²²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3076>

²²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3074>

²²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3073>

²²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3072>

²²⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3071>

²²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3070>

²²⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3069>

²²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3068>

²²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3067>

²²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3066>

²²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3065>

²²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3064>

- PR #3062²²⁹⁹ - Fix exception handling for execution::seq
- PR #3061²³⁰⁰ - Adapt MSVC C++ mode handling to VS15.5
- PR #3060²³⁰¹ - Fix compiler problem in MSVC release mode
- PR #3059²³⁰² - Fixing #2931
- Issue #3058²³⁰³ - minimal_timed_async_executor_test_exe fails to compile on master (d6f505c)
- PR #3057²³⁰⁴ - Fix stable_merge_2964 compilation problems
- PR #3056²³⁰⁵ - Fix some build warnings caused by unused variables/unnecessary tests
- PR #3055²³⁰⁶ - Update documentation for running tests
- Issue #3054²³⁰⁷ - Assertion failure when using bulk hpx::new_ in asynchronous mode
- PR #3052²³⁰⁸ - Do not bind test running to cmake test build rule
- PR #3051²³⁰⁹ - Fix HPX-Qt interaction in Qt example.
- Issue #3048²³¹⁰ - nqueen example fails occasionally
- PR #3047²³¹¹ - Fixing #3044
- PR #3046²³¹² - Add OS thread suspension
- PR #3042²³¹³ - PyCicle - first attempt at a build tool for checking PR's
- PR #3041²³¹⁴ - Fix a problem about asynchronous execution of parallel::merge and parallel::partition.
- PR #3040²³¹⁵ - Fix a mistake about exception handling in asynchronous execution of scan_partitioner.
- PR #3039²³¹⁶ - Consistently use executors to schedule work
- PR #3038²³¹⁷ - Fixing local direct function execution and lambda actions perfect forwarding
- PR #3035²³¹⁸ - Make parallel unit test names match build target/folder names
- PR #3033²³¹⁹ - Fix setting of default build type
- Issue #3032²³²⁰ - Fix partitioner arg copy found in #2982
- Issue #3031²³²¹ - Errors linking libhpx.so due to missing references (master branch, commit 6679a8882)

²²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3062>

²³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3061>

²³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3060>

²³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3059>

²³⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/3058>

²³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3057>

²³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3056>

²³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3055>

²³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3054>

²³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3052>

²³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3051>

²³¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3048>

²³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3047>

²³¹² <https://github.com/STELLAR-GROUP/hpx/pull/3046>

²³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3042>

²³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3041>

²³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3040>

²³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3039>

²³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3038>

²³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3035>

²³¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3033>

²³²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3032>

²³²¹ <https://github.com/STELLAR-GROUP/hpx/issues/3031>

- PR #3030²³²² - Revert “implement executor then interface with && forwarding reference”
- PR #3029²³²³ - Run CI inspect checks before building
- PR #3028²³²⁴ - Added range version of parallel::move
- Issue #3027²³²⁵ - Implement all scheduling APIs in terms of executors
- PR #3026²³²⁶ - implement executor then interface with && forwarding reference
- PR #3025²³²⁷ - Fix typo uninitialized to uninitialized
- PR #3024²³²⁸ - Inspect fixes
- PR #3023²³²⁹ - P0356 Simplified partial function application
- PR #3022²³³⁰ - Master fixes
- PR #3021²³³¹ - Segfault fix
- PR #3020²³³² - Disable command-line aliasing for applications that use user_main
- PR #3019²³³³ - Adding enable_elasticity option to pool configuration
- PR #3018²³³⁴ - Fix stack overflow detection configuration in header files
- PR #3017²³³⁵ - Speed up local action execution
- PR #3016²³³⁶ - Unify stack-overflow detection options, remove reference to libsigsegv
- PR #3015²³³⁷ - Speeding up accessing the resource partitioner and the topology info
- Issue #3014²³³⁸ - HPX does not compile on POWER8 with gcc 5.4
- Issue #3013²³³⁹ - hello_world occasionally prints multiple lines from a single OS-thread
- PR #3012²³⁴⁰ - Silence warning about casting away qualifiers in itt_notify.hpp
- PR #3011²³⁴¹ - Fix cpuset leak in hwloc_topology_info.cpp
- PR #3010²³⁴² - Remove useless decay_copy
- PR #3009²³⁴³ - Fixing 2996
- PR #3008²³⁴⁴ - Remove unused internal function

2322 <https://github.com/STELLAR-GROUP/hpx/pull/3030>

2323 <https://github.com/STELLAR-GROUP/hpx/pull/3029>

2324 <https://github.com/STELLAR-GROUP/hpx/pull/3028>

2325 <https://github.com/STELLAR-GROUP/hpx/issues/3027>

2326 <https://github.com/STELLAR-GROUP/hpx/pull/3026>

2327 <https://github.com/STELLAR-GROUP/hpx/pull/3025>

2328 <https://github.com/STELLAR-GROUP/hpx/pull/3024>

2329 <https://github.com/STELLAR-GROUP/hpx/pull/3023>

2330 <https://github.com/STELLAR-GROUP/hpx/pull/3022>

2331 <https://github.com/STELLAR-GROUP/hpx/pull/3021>

2332 <https://github.com/STELLAR-GROUP/hpx/pull/3020>

2333 <https://github.com/STELLAR-GROUP/hpx/pull/3019>

2334 <https://github.com/STELLAR-GROUP/hpx/pull/3018>

2335 <https://github.com/STELLAR-GROUP/hpx/pull/3017>

2336 <https://github.com/STELLAR-GROUP/hpx/pull/3016>

2337 <https://github.com/STELLAR-GROUP/hpx/pull/3015>

2338 <https://github.com/STELLAR-GROUP/hpx/issues/3014>

2339 <https://github.com/STELLAR-GROUP/hpx/issues/3013>

2340 <https://github.com/STELLAR-GROUP/hpx/pull/3012>

2341 <https://github.com/STELLAR-GROUP/hpx/pull/3011>

2342 <https://github.com/STELLAR-GROUP/hpx/pull/3010>

2343 <https://github.com/STELLAR-GROUP/hpx/pull/3009>

2344 <https://github.com/STELLAR-GROUP/hpx/pull/3008>

- PR #3007²³⁴⁵ - Fixing wrapper_heap alignment problems
- Issue #3006²³⁴⁶ - hwloc memory leak
- PR #3004²³⁴⁷ - Silence C4251 (needs to have dll-interface) for future_data_void
- Issue #3003²³⁴⁸ - Suspension of runtime
- PR #3001²³⁴⁹ - Attempting to avoid data races in async_traversal while evaluating dataflow()
- PR #3000²³⁵⁰ - Adding hpx::util::optional as a first step to replace experimental::optional
- PR #2998²³⁵¹ - Cleanup up and Fixing component creation and deletion
- Issue #2996²³⁵² - Build fails with HPX_WITH_HWLOC=OFF
- PR #2995²³⁵³ - Push more future_data functionality to source file
- PR #2994²³⁵⁴ - WIP: Fix throttle test
- PR #2993²³⁵⁵ - Making sure -hpx:help does not throw for required (but missing) arguments
- PR #2992²³⁵⁶ - Adding non-blocking (on destruction) service executors
- Issue #2991²³⁵⁷ - run_as_os_thread locks up
- Issue #2990²³⁵⁸ - --help will not work until all required options are provided
- PR #2989²³⁵⁹ - Improve error messages caused by misuse of dataflow
- PR #2988²³⁶⁰ - Improve error messages caused by misuse of .then
- Issue #2987²³⁶¹ - stack overflow detection producing false positives
- PR #2986²³⁶² - Deduplicate non-dependent thread_info logging types
- PR #2985²³⁶³ - Adapted parallel::{all_oflany_oflnone_of} for Ranges TS (see #1668)
- PR #2984²³⁶⁴ - Refactor one_size_heap code to simplify code
- PR #2983²³⁶⁵ - Fixing local_new_component
- PR #2982²³⁶⁶ - Clang tidy
- PR #2981²³⁶⁷ - Simplify allocator rebinding in pack traversal

²³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3007>

²³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3006>

²³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3004>

²³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3003>

²³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3001>

²³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3000>

²³⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2998>

²³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2996>

²³⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2995>

²³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2994>

²³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2993>

²³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2992>

²³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2991>

²³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2990>

²³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2989>

²³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2988>

²³⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2987>

²³⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2986>

²³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2985>

²³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2984>

²³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2983>

²³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2982>

²³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2981>

- PR #2979²³⁶⁸ - Fixing integer overflows
- PR #2978²³⁶⁹ - Implement parallel::inplace_merge
- Issue #2977²³⁷⁰ - Make hwloc compulsory instead of optional
- PR #2976²³⁷¹ - Making sure client_base instance that registered the component does not unregister it when being destructed
- PR #2975²³⁷² - Change version of pulled APEX to master
- PR #2974²³⁷³ - Fix domain not being freed at the end of scheduling loop
- PR #2973²³⁷⁴ - Fix small typos
- PR #2972²³⁷⁵ - Adding uintstd.h header
- PR #2971²³⁷⁶ - Fall back to creating local components using local_new
- PR #2970²³⁷⁷ - Improve is_tuple_like trait
- PR #2969²³⁷⁸ - Fix HPX_WITH_MORE_THAN_64_THREADS default value
- PR #2968²³⁷⁹ - Cleaning up dataflow overload set
- PR #2967²³⁸⁰ - Make parallel::merge is stable. (Fix #2964.)
- PR #2966²³⁸¹ - Fixing a couple of held locks during exception handling
- PR #2965²³⁸² - Adding missing #include
- Issue #2964²³⁸³ - parallel merge is not stable
- PR #2963²³⁸⁴ - Making sure any function object passed to dataflow is released after being invoked
- PR #2962²³⁸⁵ - Partially reverting #2891
- PR #2961²³⁸⁶ - Attempt to fix the gcc 4.9 problem with the async pack traversal
- Issue #2959²³⁸⁷ - Program terminates during error handling
- Issue #2958²³⁸⁸ - HPX_PLAIN_ACTION breaks due to missing include
- PR #2957²³⁸⁹ - Fixing errors generated by mixing different attribute syntaxes
- Issue #2956²³⁹⁰ - Mixing attribute syntaxes leads to compiler errors

²³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2979>

²³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2978>

²³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2977>

²³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2976>

²³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2975>

²³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2974>

²³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2973>

²³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2972>

²³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2971>

²³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2970>

²³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2969>

²³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2968>

²³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2967>

²³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2966>

²³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2965>

²³⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/2964>

²³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2963>

²³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2962>

²³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2961>

²³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2959>

²³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2958>

²³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2957>

²³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2956>

- Issue #2955²³⁹¹ - Fix OS-Thread throttling
- PR #2953²³⁹² - Making sure any hpx.os_threads=N supplied through a –hpx::config file is taken into account
- PR #2952²³⁹³ - Removing wrong call to cleanup_terminated_locked
- PR #2951²³⁹⁴ - Revert “Make sure the function vtables are initialized before use”
- PR #2950²³⁹⁵ - Fix a namespace compilation error when some schedulers are disabled
- Issue #2949²³⁹⁶ - master branch giving lockups on shutdown
- Issue #2947²³⁹⁷ - hpx.ini is not used correctly at initialization
- PR #2946²³⁹⁸ - Adding explicit feature test for thread_local
- PR #2945²³⁹⁹ - Make sure the function vtables are initialized before use
- PR #2944²⁴⁰⁰ - Attempting to solve affinity problems on CircleCI
- PR #2943²⁴⁰¹ - Changing channel actions to be direct
- PR #2942²⁴⁰² - Adding split_future for std::vector
- PR #2941²⁴⁰³ - Add a feature test to test for CXX11 override
- Issue #2940²⁴⁰⁴ - Add split_future for future<vector<T>>
- PR #2939²⁴⁰⁵ - Making error reporting during problems with setting affinity masks more verbose
- PR #2938²⁴⁰⁶ - Fix this various executors
- PR #2937²⁴⁰⁷ - Fix some typos in documentation
- PR #2934²⁴⁰⁸ - Remove the need for “complete” SFINAE checks
- PR #2933²⁴⁰⁹ - Making sure parallel::for_loop is executed in parallel if requested
- PR #2932²⁴¹⁰ - Classify chunk_size_iterator to input iterator tag. (Fix #2866)
- Issue #2931²⁴¹¹ - –hpx:help triggers unusual error with clang build
- PR #2930²⁴¹² - Add #include files needed to set _POSIX_VERSION for debug check
- PR #2929²⁴¹³ - Fix a couple of deprecated c++ features

²³⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2955>

²³⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2953>

²³⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2952>

²³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2951>

²³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2950>

²³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2949>

²³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2947>

²³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2946>

²³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2945>

²⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2944>

²⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2943>

²⁴⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2942>

²⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2941>

²⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2940>

²⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2939>

²⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2938>

²⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2937>

²⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2934>

²⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2933>

²⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2932>

²⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2931>

²⁴¹² <https://github.com/STELLAR-GROUP/hpx/pull/2930>

²⁴¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2929>

- PR #2928²⁴¹⁴ - Fixing execution parameters
- Issue #2927²⁴¹⁵ - CMake warning: ... cycle in constraint graph
- PR #2926²⁴¹⁶ - Default pool rename
- Issue #2925²⁴¹⁷ - Default pool cannot be renamed
- Issue #2924²⁴¹⁸ - hpx:attach-debugger=startup does not work any more
- PR #2923²⁴¹⁹ - Alloc membind
- PR #2922²⁴²⁰ - This fixes CircleCI errors when running with -hpx:bind=none
- PR #2921²⁴²¹ - Custom pool executor was missing priority and stacksize options
- PR #2920²⁴²² - Adding test to trigger problem reported in #2916
- PR #2919²⁴²³ - Make sure the resource_partitioner is properly destructed on hpx::finalize
- Issue #2918²⁴²⁴ - hpx::init calls wrong (first) callback when called multiple times
- PR #2917²⁴²⁵ - Adding util::checkpoint
- Issue #2916²⁴²⁶ - Weird runtime failures when using a channel and chained continuations
- PR #2915²⁴²⁷ - Introduce executor parameters customization points
- Issue #2914²⁴²⁸ - Task assignment to current Pool has unintended consequences
- PR #2913²⁴²⁹ - Fix rp hang
- PR #2912²⁴³⁰ - Update contributors
- PR #2911²⁴³¹ - Fixing CUDA problems
- PR #2910²⁴³² - Improve error reporting for process component on POSIX systems
- PR #2909²⁴³³ - Fix typo in include path
- PR #2908²⁴³⁴ - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2907²⁴³⁵ - Optionally force-delete remaining channel items on close
- PR #2906²⁴³⁶ - Making sure generated performance counter names are correct

²⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2928>

²⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2927>

²⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2926>

²⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2925>

²⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2924>

²⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2923>

²⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2922>

²⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2921>

²⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/2920>

²⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/2919>

²⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2918>

²⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2917>

²⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2916>

²⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2915>

2428 <https://github.com/STELLAR-GROUP/hpx/issues/2914>2429 <https://github.com/STELLAR-GROUP/hpx/pull/2913>2430 <https://github.com/STELLAR-GROUP/hpx/pull/2912>2431 <https://github.com/STELLAR-GROUP/hpx/pull/2911>2432 <https://github.com/STELLAR-GROUP/hpx/pull/2910>2433 <https://github.com/STELLAR-GROUP/hpx/pull/2909>2434 <https://github.com/STELLAR-GROUP/hpx/pull/2908>2435 <https://github.com/STELLAR-GROUP/hpx/pull/2907>2436 <https://github.com/STELLAR-GROUP/hpx/pull/2906>

- Issue #2905²⁴³⁷ - collecting idle-rate performance counters on multiple localities produces an error
- Issue #2904²⁴³⁸ - build broken for Intel 17 compilers
- PR #2903²⁴³⁹ - Documentation Updates– Adding New People
- PR #2902²⁴⁴⁰ - Fixing service_executor
- PR #2901²⁴⁴¹ - Fixing partitioned_vector creation
- PR #2900²⁴⁴² - Add numa-balanced mode to hpx::bind, spread cores over numa domains
- Issue #2899²⁴⁴³ - hpx::bind does not have a mode that balances cores over numa domains
- PR #2898²⁴⁴⁴ - Adding missing #include and missing guard for optional code section
- PR #2897²⁴⁴⁵ - Removing dependency on Boost. ICL
- Issue #2896²⁴⁴⁶ - Debug build fails without -fpermissive with GCC 7.1 and Boost 1.65
- PR #2895²⁴⁴⁷ - Fixing SLURM environment parsing
- PR #2894²⁴⁴⁸ - Fix incorrect handling of compile definition with value 0
- Issue #2893²⁴⁴⁹ - Disabling schedulers causes build errors
- PR #2892²⁴⁵⁰ - added list serializer
- PR #2891²⁴⁵¹ - Resource Partitioner Fixes
- Issue #2890²⁴⁵² - Destroying a non-empty channel causes an assertion failure
- PR #2889²⁴⁵³ - Add check for libatomic
- PR #2888²⁴⁵⁴ - Fix compilation problems if HPX_WITH_ITT_NOTIFY=ON
- PR #2887²⁴⁵⁵ - Adapt broadcast() to non-unwrapping async<Action>
- PR #2886²⁴⁵⁶ - Replace Boost.Random with C++11 <random>
- Issue #2885²⁴⁵⁷ - regression in broadcast?
- Issue #2884²⁴⁵⁸ - linking -latomic is not portable
- PR #2883²⁴⁵⁹ - Explicitly set -pthread flag if available

²⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2905>

²⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2904>

²⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2903>

²⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2902>

²⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2901>

²⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2900>

²⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2899>

²⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2898>

²⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2897>

²⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2896>

²⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2895>

²⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2894>

²⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2893>

²⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2892>

²⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2891>

²⁴⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2890>

²⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2889>

²⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2888>

²⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2887>

²⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2886>

²⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2885>

²⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2884>

²⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2883>

- PR #2882²⁴⁶⁰ - Wrap boost::format uses
- Issue #2881²⁴⁶¹ - hpx not compiling with HPX_WITH_ITTNOTIFY=On
- Issue #2880²⁴⁶² - hpx::bind scatter/balanced give wrong pu masks
- PR #2878²⁴⁶³ - Fix incorrect pool usage masks setup in RP/thread manager
- PR #2877²⁴⁶⁴ - Require std::array by default
- PR #2875²⁴⁶⁵ - Deprecate use of BOOST_ASSERT
- PR #2874²⁴⁶⁶ - Changed serialization of boost.variant to use variadic templates
- Issue #2873²⁴⁶⁷ - building with parcelport_mpi fails on cori
- PR #2871²⁴⁶⁸ - Adding missing support for throttling scheduler
- PR #2870²⁴⁶⁹ - Disambiguate use of base_lco_with_value macros with channel
- Issue #2869²⁴⁷⁰ - Difficulty compiling HPX_REGISTER_CHANNEL_DECLARATION(double)
- PR #2868²⁴⁷¹ - Removing unneeded assert
- PR #2867²⁴⁷² - Implement parallel::unique
- Issue #2866²⁴⁷³ - The chunk_size_iterator violates multipass guarantee
- PR #2865²⁴⁷⁴ - Only use sched_getcpu on linux machines
- PR #2864²⁴⁷⁵ - Create redistribution archive for successful builds
- PR #2863²⁴⁷⁶ - Replace casts/assignments with hard-coded memcpy operations
- Issue #2862²⁴⁷⁷ - sched_getcpu not available on MacOS
- PR #2861²⁴⁷⁸ - Fixing unmatched header defines and recursive inclusion of threadmanager
- Issue #2860²⁴⁷⁹ - Master program fails with assertion ‘type == data_type_address’ failed: HPX(assertion_failure)
- Issue #2852²⁴⁸⁰ - Support for ARM64
- PR #2858²⁴⁸¹ - Fix misplaced #if #endif's that cause build failure without THREAD_CUMULATIVE_COUNTS

²⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2882>
²⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2881>
²⁴⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2880>
²⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2878>
²⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2877>
²⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2875>
²⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2874>
²⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2873>
²⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2871>
²⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2870>
²⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2869>
²⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2868>
²⁴⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2867>
²⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/2866>
²⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2865>
²⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2864>
²⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2863>
²⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2862>
²⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2861>
²⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2860>
²⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2852>
²⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2858>

- PR #2857²⁴⁸² - Fix some listing in documentation
- PR #2856²⁴⁸³ - Fixing component handling for lcos
- PR #2855²⁴⁸⁴ - Add documentation for coarrays
- PR #2854²⁴⁸⁵ - Support ARM64 in timestamps
- PR #2853²⁴⁸⁶ - Update Table 17. Non-modifying Parallel Algorithms in Documentation
- PR #2851²⁴⁸⁷ - Allowing for non-default-constructible component types
- PR #2850²⁴⁸⁸ - Enable returning future<R> from actions where R is not default-constructible
- PR #2849²⁴⁸⁹ - Unify serialization of non-default-constructable types
- Issue #2848²⁴⁹⁰ - Components have to be default constructible
- Issue #2847²⁴⁹¹ - Returning a future<R> where R is not default-constructable broken
- Issue #2846²⁴⁹² - Unify serialization of non-default-constructible types
- PR #2845²⁴⁹³ - Add Visual Studio 2015 to the tested toolchains in Appveyor
- Issue #2844²⁴⁹⁴ - Change the appveyor build to use the minimal required MSVC version
- Issue #2843²⁴⁹⁵ - multi node hello_world hangs
- PR #2842²⁴⁹⁶ - Correcting Spelling mistake in docs
- PR #2841²⁴⁹⁷ - Fix usage of std::aligned_storage
- PR #2840²⁴⁹⁸ - Remove constexpr from a void function
- Issue #2839²⁴⁹⁹ - memcpy buffer overflow: load_construct_data() and std::complex members
- Issue #2835²⁵⁰⁰ - constexpr functions with void return type break compilation with CUDA 8.0
- Issue #2834²⁵⁰¹ - One suspicion in parallel::detail::handle_exception
- PR #2833²⁵⁰² - Implement parallel::merge
- PR #2832²⁵⁰³ - Fix a strange thing in parallel::util::detail::handle_local_exceptions. (Fix #2818)
- PR #2830²⁵⁰⁴ - Break the debugger when a test failed

²⁴⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2857>

²⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2856>

²⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2855>

²⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2854>

²⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2853>

²⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2851>

²⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2850>

²⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2849>

²⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2848>

²⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2847>

²⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/2846>

²⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2845>

²⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2844>

²⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2843>

²⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2842>

²⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2841>

²⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2840>

²⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2839>

²⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2835>

²⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/2834>

²⁵⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2833>

²⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2832>

²⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2830>

- Issue #2831²⁵⁰⁵ - parallel/executors/execution_fwd.hpp causes compilation failure in C++11 mode.
- PR #2829²⁵⁰⁶ - Implement an API for asynchronous pack traversal
- PR #2828²⁵⁰⁷ - Split unit test builds on CircleCI to avoid timeouts
- Issue #2827²⁵⁰⁸ - failure to compile hello_world example with -Werror
- PR #2824²⁵⁰⁹ - Making sure promises are marked as started when used as continuations
- PR #2823²⁵¹⁰ - Add documentation for partitioned_vector_view
- Issue #2822²⁵¹¹ - Yet another issue with wait_for similar to #2796
- PR #2821²⁵¹² - Fix bugs and improve that about HPX_HAVE_CXX11_AUTO_RETURN_VALUE of CMake
- PR #2820²⁵¹³ - Support C++11 in benchmark codes of parallel::partition and parallel::partition_copy
- PR #2819²⁵¹⁴ - Fix compile errors in unit test of container version of parallel::partition
- Issue #2818²⁵¹⁵ - A strange thing in parallel::util::detail::handle_local_exceptions
- Issue #2815²⁵¹⁶ - HPX fails to compile with HPX_WITH_CUDA=ON and the new CUDA 9.0 RC
- Issue #2814²⁵¹⁷ - Using ‘gmakeN’ after ‘cmake’ produces error in src/CMakeFiles/hpx.dir/runtime/agas/addressing_service.cpp.o
- PR #2813²⁵¹⁸ - Properly support [[noreturn]] attribute if available
- Issue #2812²⁵¹⁹ - Compilation fails with gcc 7.1.1
- PR #2811²⁵²⁰ - Adding hpx::launch::lazy and support for async, dataflow, and future::then
- PR #2810²⁵²¹ - Add option allowing to disable deprecation warning
- PR #2809²⁵²² - Disable throttling scheduler if HWLOC is not found/used
- PR #2808²⁵²³ - Fix compile errors on some environments of parallel::partition
- Issue #2807²⁵²⁴ - Difficulty building with HPX_WITH_HWLOC=Off
- PR #2806²⁵²⁵ - Partitioned vector
- PR #2805²⁵²⁶ - Serializing collections with non-default constructible data

²⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2831>

²⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2829>

²⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2828>

²⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2827>

²⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2824>

²⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2823>

²⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2822>

²⁵¹² <https://github.com/STELLAR-GROUP/hpx/pull/2821>

²⁵¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2820>

²⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2819>

²⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2818>

²⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2815>

²⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2814>

²⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2813>

²⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2812>

²⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2811>

²⁵²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2810>

²⁵²² <https://github.com/STELLAR-GROUP/hpx/pull/2809>

²⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/2808>

²⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2807>

²⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2806>

²⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2805>

- PR #2802²⁵²⁷ - Fix FreeBSD 11
- Issue #2801²⁵²⁸ - Rate limiting techniques in io_service
- Issue #2800²⁵²⁹ - New Launch Policy: async_if
- PR #2799²⁵³⁰ - Fix a unit test failure on GCC in tuple_cat
- PR #2798²⁵³¹ - bump minimum required cmake to 3.0 in test
- PR #2797²⁵³² - Making sure future::wait_for et.al. work properly for action results
- Issue #2796²⁵³³ - wait_for does always in “deferred” state for calls on remote localities
- Issue #2795²⁵³⁴ - Serialization of types without default constructor
- PR #2794²⁵³⁵ - Fixing test for partitioned_vector iteration
- PR #2792²⁵³⁶ - Implemented segmented find and its variations for partitioned vector
- PR #2791²⁵³⁷ - Circumvent scary warning about placement new
- PR #2790²⁵³⁸ - Fix OSX build
- PR #2789²⁵³⁹ - Resource partitioner
- PR #2788²⁵⁴⁰ - Adapt parallel::is_heap and parallel::is_heap_until to Ranges TS
- PR #2787²⁵⁴¹ - Unwrap hotfixes
- PR #2786²⁵⁴² - Update CMake Minimum Version to 3.3.2 (refs #2565)
- Issue #2785²⁵⁴³ - Issues with masks and cpuset
- PR #2784²⁵⁴⁴ - Error with reduce and transform reduce fixed
- PR #2783²⁵⁴⁵ - StackOverflow integration with libsigsegv
- PR #2782²⁵⁴⁶ - Replace boost::atomic with std::atomic (where possible)
- PR #2781²⁵⁴⁷ - Check for and optionally use [[deprecated]] attribute
- PR #2780²⁵⁴⁸ - Adding empty (but non-trivial) destructor to circumvent warnings
- PR #2779²⁵⁴⁹ - Exception info tweaks

²⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2802>

²⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2801>

²⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2800>

²⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2799>

²⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2798>

²⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/2797>

²⁵³³ <https://github.com/STELLAR-GROUP/hpx/issues/2796>

²⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2795>

²⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2794>

²⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2792>

²⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2791>

²⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2790>

²⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2789>

²⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2788>

²⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2787>

²⁵⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2786>

²⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2785>

²⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2784>

²⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2783>

²⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2782>

²⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2781>

²⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2780>

²⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2779>

- PR #2778²⁵⁵⁰ - Implement parallel::partition
- PR #2777²⁵⁵¹ - Improve error handling in gather_here/gather_there
- PR #2776²⁵⁵² - Fix a bug in compiler version check
- PR #2775²⁵⁵³ - Fix compilation when HPX_WITH_LOGGING is OFF
- PR #2774²⁵⁵⁴ - Removing dependency on Boost.Date_Time
- PR #2773²⁵⁵⁵ - Add sync_images() method to spmd_block class
- PR #2772²⁵⁵⁶ - Adding documentation for PAPI counters
- PR #2771²⁵⁵⁷ - Removing boost preprocessor dependency
- PR #2770²⁵⁵⁸ - Adding test, fixing deadlock in config registry
- PR #2769²⁵⁵⁹ - Remove some other warnings and errors detected by clang 5.0
- Issue #2768²⁵⁶⁰ - Is there iterator tag for HPX?
- PR #2767²⁵⁶¹ - Improvements to continuation annotation
- PR #2765²⁵⁶² - gcc split stack support for HPX threads #620
- PR #2764²⁵⁶³ - Fix some uses of begin/end, remove unnecessary includes
- PR #2763²⁵⁶⁴ - Bump minimal Boost version to 1.55.0
- PR #2762²⁵⁶⁵ - hpx::partitioned_vector serializer
- PR #2761²⁵⁶⁶ - Adding configuration summary to cmake output and -hpx:info
- PR #2760²⁵⁶⁷ - Removing 1d_hydro example as it is broken
- PR #2758²⁵⁶⁸ - Remove various warnings detected by clang 5.0
- Issue #2757²⁵⁶⁹ - In case of a “raw thread” is needed per core for implementing parallel algorithm, what is good practice in HPX?
- PR #2756²⁵⁷⁰ - Allowing for LCOs to be simple components
- PR #2755²⁵⁷¹ - Removing make_index_pack_unrolled
- PR #2754²⁵⁷² - Implement parallel::unique_copy

2550 <https://github.com/STELLAR-GROUP/hpx/pull/2778>
2551 <https://github.com/STELLAR-GROUP/hpx/pull/2777>
2552 <https://github.com/STELLAR-GROUP/hpx/pull/2776>
2553 <https://github.com/STELLAR-GROUP/hpx/pull/2775>
2554 <https://github.com/STELLAR-GROUP/hpx/pull/2774>
2555 <https://github.com/STELLAR-GROUP/hpx/pull/2773>
2556 <https://github.com/STELLAR-GROUP/hpx/pull/2772>
2557 <https://github.com/STELLAR-GROUP/hpx/pull/2771>
2558 <https://github.com/STELLAR-GROUP/hpx/pull/2770>
2559 <https://github.com/STELLAR-GROUP/hpx/pull/2769>
2560 <https://github.com/STELLAR-GROUP/hpx/issues/2768>
2561 <https://github.com/STELLAR-GROUP/hpx/pull/2767>
2562 <https://github.com/STELLAR-GROUP/hpx/pull/2765>
2563 <https://github.com/STELLAR-GROUP/hpx/pull/2764>
2564 <https://github.com/STELLAR-GROUP/hpx/pull/2763>
2565 <https://github.com/STELLAR-GROUP/hpx/pull/2762>
2566 <https://github.com/STELLAR-GROUP/hpx/pull/2761>
2567 <https://github.com/STELLAR-GROUP/hpx/pull/2760>
2568 <https://github.com/STELLAR-GROUP/hpx/pull/2758>
2569 <https://github.com/STELLAR-GROUP/hpx/issues/2757>
2570 <https://github.com/STELLAR-GROUP/hpx/pull/2756>
2571 <https://github.com/STELLAR-GROUP/hpx/pull/2755>
2572 <https://github.com/STELLAR-GROUP/hpx/pull/2754>

- PR #2753²⁵⁷³ - Fixing detection of [[fallthrough]] attribute
- PR #2752²⁵⁷⁴ - New thread priority names
- PR #2751²⁵⁷⁵ - Replace boost::exception with proposed exception_info
- PR #2750²⁵⁷⁶ - Replace boost::iterator_range
- PR #2749²⁵⁷⁷ - Fixing hdf5 examples
- Issue #2748²⁵⁷⁸ - HPX fails to build with enabled hdf5 examples
- Issue #2747²⁵⁷⁹ - Inherited task priorities break certain DAG optimizations
- Issue #2746²⁵⁸⁰ - HPX segfaulting with valgrind
- PR #2745²⁵⁸¹ - Adding extended arithmetic performance counters
- PR #2744²⁵⁸² - Adding ability to statistics counters to reset base counter
- Issue #2743²⁵⁸³ - Statistics counter does not support resetting
- PR #2742²⁵⁸⁴ - Making sure Vc V2 builds without additional HPX configuration flags
- PR #2741²⁵⁸⁵ - Deprecate unwrapped and implement unwrap and unwrapping
- PR #2740²⁵⁸⁶ - Coroutine stackoverflow detection for linux/posix; Issue #2408
- PR #2739²⁵⁸⁷ - Add files via upload
- PR #2738²⁵⁸⁸ - Appveyor support
- PR #2737²⁵⁸⁹ - Fixing 2735
- Issue #2736²⁵⁹⁰ - 1d_hydro example doesn't work
- Issue #2735²⁵⁹¹ - partitioned_vector_subview test failing
- PR #2734²⁵⁹² - Add C++11 range utilities
- PR #2733²⁵⁹³ - Adapting iterator requirements for parallel algorithms
- PR #2732²⁵⁹⁴ - Integrate C++ Co-arrays
- PR #2731²⁵⁹⁵ - Adding on_migrated event handler to migratable component instances

²⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2753>

²⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2752>

²⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2751>

²⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2750>

²⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2749>

²⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2748>

²⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2747>

²⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2746>

²⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2745>

²⁵⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2744>

²⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/2743>

²⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2742>

²⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2741>

²⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2740>

²⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2739>

²⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2738>

²⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2737>

²⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2736>

²⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2735>

²⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2734>

²⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2733>

²⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2732>

²⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2731>

- Issue #2729²⁵⁹⁶ - Add on_migrated() event handler to migratable components
- Issue #2728²⁵⁹⁷ - Why Projection is needed in parallel algorithms?
- PR #2727²⁵⁹⁸ - Cmake files for StackOverflow Detection
- PR #2726²⁵⁹⁹ - CMake for Stack Overflow Detection
- PR #2725²⁶⁰⁰ - Implemented segmented algorithms for partitioned vector
- PR #2724²⁶⁰¹ - Fix examples in Action documentation
- PR #2723²⁶⁰² - Enable lcos::channel<T>::register_as
- Issue #2722²⁶⁰³ - channel register_as() failing on compilation
- PR #2721²⁶⁰⁴ - Mind map
- PR #2720²⁶⁰⁵ - reorder forward declarations to get rid of C++14-only auto return types
- PR #2719²⁶⁰⁶ - Add documentation for partitioned_vector and add features in pack.hpp
- Issue #2718²⁶⁰⁷ - Some forward declarations in execution_fwd.hpp aren't C++11-compatible
- PR #2717²⁶⁰⁸ - Config support for fallthrough attribute
- PR #2716²⁶⁰⁹ - Implement parallel::partition_copy
- PR #2715²⁶¹⁰ - initial import of icu string serializer
- PR #2714²⁶¹¹ - initial import of valarray serializer
- PR #2713²⁶¹² - Remove slashes before CMAKE_FILES_DIRECTORY variables
- PR #2712²⁶¹³ - Fixing wait for 1751
- PR #2711²⁶¹⁴ - Adjust code for minimal supported GCC having been bumped to 4.9
- PR #2710²⁶¹⁵ - Adding code of conduct
- PR #2709²⁶¹⁶ - Fixing UB in destroy tests
- PR #2708²⁶¹⁷ - Add inline to prevent multiple definition issue
- Issue #2707²⁶¹⁸ - Multiple defined symbols for task_block.hpp in VS2015

²⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2729>

²⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2728>

²⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2727>

²⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2726>

²⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2725>

²⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2724>

²⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2723>

²⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2722>

²⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2721>

²⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2720>

²⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2719>

²⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2718>

²⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2717>

²⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2716>

²⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2715>

²⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2714>

²⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/2713>

²⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2712>

²⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2711>

²⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2710>

²⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2709>

²⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2708>

²⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2707>

- PR #2706²⁶¹⁹ - Adding .clang-format file
- PR #2704²⁶²⁰ - Add a synchronous mapping API
- Issue #2703²⁶²¹ - Request: Add the .clang-format file to the repository
- Issue #2702²⁶²² - STELLAR-GROUP/Vc slower than VCv1 possibly due to wrong instructions generated
- Issue #2701²⁶²³ - Datapar with STELLAR-GROUP/Vc requires obscure flag
- Issue #2700²⁶²⁴ - Naming inconsistency in parallel algorithms
- Issue #2699²⁶²⁵ - Iterator requirements are different from standard in parallel copy_if.
- PR #2698²⁶²⁶ - Properly releasing parcelport write handlers
- Issue #2697²⁶²⁷ - Compile error in addressing_service.cpp
- Issue #2696²⁶²⁸ - Building and using HPX statically: undefined references from runtime_support_server.cpp
- Issue #2695²⁶²⁹ - Executor changes cause compilation failures
- PR #2694²⁶³⁰ - Refining C++ language mode detection for MSVC
- PR #2693²⁶³¹ - P0443 r2
- PR #2692²⁶³² - Partially reverting changes to parcel_await
- Issue #2689²⁶³³ - HPX build fails when HPX_WITH_CUDA is enabled
- PR #2688²⁶³⁴ - Make Cuda Clang builds pass
- PR #2687²⁶³⁵ - Add an is_tuple_like trait for sequenceable type detection
- PR #2686²⁶³⁶ - Allowing throttling scheduler to be used without idle backoff
- PR #2685²⁶³⁷ - Add support of std::array to hpx::util::tuple_size and tuple_element
- PR #2684²⁶³⁸ - Adding new statistics performance counters
- PR #2683²⁶³⁹ - Replace boost::exception_ptr with std::exception_ptr
- Issue #2682²⁶⁴⁰ - HPX does not compile with HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF
- PR #2681²⁶⁴¹ - Attempt to fix problem in managed_component_base

²⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2706>

²⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2704>

²⁶²¹ <https://github.com/STELLAR-GROUP/hpx/issues/2703>

²⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/2702>

²⁶²³ <https://github.com/STELLAR-GROUP/hpx/issues/2701>

²⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2700>

²⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2699>

²⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2698>

²⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2697>

²⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2696>

²⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2695>

²⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2694>

²⁶³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2693>

²⁶³² <https://github.com/STELLAR-GROUP/hpx/pull/2692>

²⁶³³ <https://github.com/STELLAR-GROUP/hpx/issues/2689>

²⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2688>

²⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2687>

²⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2686>

²⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2685>

²⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2684>

²⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2683>

²⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2682>

²⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2681>

- PR #2680²⁶⁴² - Fix bad size during archive creation
- Issue #2679²⁶⁴³ - Mismatch between size of archive and container
- Issue #2678²⁶⁴⁴ - In parallel algorithm, other tasks are executed to the end even if an exception occurs in any task.
- PR #2677²⁶⁴⁵ - Adding include check for std::addressof
- PR #2676²⁶⁴⁶ - Adding parallel::destroy and destroy_n
- PR #2675²⁶⁴⁷ - Making sure statistics counters work as expected
- PR #2674²⁶⁴⁸ - Turning assertions into exceptions
- PR #2673²⁶⁴⁹ - Inhibit direct conversion from future<future<T>> -> future<void>
- PR #2672²⁶⁵⁰ - C++17 invoke forms
- PR #2671²⁶⁵¹ - Adding uninitialized_value_construct and uninitialized_value_construct_n
- PR #2670²⁶⁵² - Integrate spmd multidimensional views for partitioned_vectors
- PR #2669²⁶⁵³ - Adding uninitialized_default_construct and uninitialized_default_construct_n
- PR #2668²⁶⁵⁴ - Fixing documentation index
- Issue #2667²⁶⁵⁵ - Ambiguity of nested hpx::future<void>'s
- Issue #2666²⁶⁵⁶ - Statistics Performance counter is not working
- PR #2664²⁶⁵⁷ - Adding uninitialized_move and uninitialized_move_n
- Issue #2663²⁶⁵⁸ - Seg fault in managed_component::get_base_gid, possibly cause by util::reinitializable_static
- Issue #2662²⁶⁵⁹ - Crash in managed_component::get_base_gid due to problem with util::reinitializable_static
- PR #2665²⁶⁶⁰ - Hide the detail namespace in doxygen per default
- PR #2660²⁶⁶¹ - Add documentation to hpx::util::unwrapped and hpx::util::unwrapped2
- PR #2659²⁶⁶² - Improve integration with vcpkg
- PR #2658²⁶⁶³ - Unify access_data trait for use in both, serialization and de-serialization
- PR #2657²⁶⁶⁴ - Removing hpx::lcos::queue<T>

²⁶⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2680>

²⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2679>

²⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2678>

²⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2677>

²⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2676>

²⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2675>

²⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2674>

²⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2673>

²⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2672>

²⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2671>

²⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2670>

²⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2669>

²⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2668>

²⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2667>

²⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2666>

²⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2664>

²⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2663>

²⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2662>

²⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2665>

²⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2660>

²⁶⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2659>

²⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2658>

²⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2657>

- PR #2656²⁶⁶⁵ - Reduce MAX_TERMINATED_THREADS default, improve memory use on manycore cpus
- PR #2655²⁶⁶⁶ - Maintenance for emulate-deleted macros
- PR #2654²⁶⁶⁷ - Implement parallel is_heap and is_heap_until
- PR #2653²⁶⁶⁸ - Drop support for VS2013
- PR #2652²⁶⁶⁹ - This patch makes sure that all parcels in a batch are properly handled
- PR #2649²⁶⁷⁰ - Update docs (Table 18) - move transform to end
- Issue #2647²⁶⁷¹ - hpx::parcelset::detail::parcel_data::has_continuation_ is uninitialized
- Issue #2644²⁶⁷² - Some .vcxproj in the HPX.sln fail to build
- Issue #2641²⁶⁷³ - hpx::lcos::queue should be deprecated
- PR #2640²⁶⁷⁴ - A new throttling policy with public APIs to suspend/resume
- PR #2639²⁶⁷⁵ - Fix a tiny typo in tutorial.
- Issue #2638²⁶⁷⁶ - Invalid return type ‘void’ of constexpr function
- PR #2636²⁶⁷⁷ - Add and use HPX_MSVC_WARNING_PRAGMA for #pragma warning
- PR #2633²⁶⁷⁸ - Distributed define_spmd_block
- PR #2632²⁶⁷⁹ - Making sure container serialization uses size-compatible types
- PR #2631²⁶⁸⁰ - Add lcos::local::one_element_channel
- PR #2629²⁶⁸¹ - Move unordered_map out of parcelport into hpx/concurrent
- PR #2628²⁶⁸² - Making sure that shutdown does not hang
- PR #2627²⁶⁸³ - Fix serialization
- PR #2626²⁶⁸⁴ - Generate cmake_variables.qbk and cmake_toolchains.qbk outside of the source tree
- PR #2625²⁶⁸⁵ - Supporting -std=c++17 flag
- PR #2624²⁶⁸⁶ - Fixing a small cmake typo
- PR #2622²⁶⁸⁷ - Update CMake minimum required version to 3.0.2 (closes #2621)

²⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2656>

²⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2655>

²⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2654>

²⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2653>

²⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2652>

²⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2649>

²⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/2647>

²⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/2644>

²⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/2641>

²⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2640>

²⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2639>

²⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2638>

²⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2636>

²⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2633>

²⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2632>

²⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2631>

²⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2629>

²⁶⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2628>

²⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2627>

²⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2626>

²⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2625>

²⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2624>

²⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2622>

- Issue #2621²⁶⁸⁸ - Compiling hpx master fails with /usr/bin/ld: final link failed: Bad value
- PR #2620²⁶⁸⁹ - Remove warnings due to some captured variables
- PR #2619²⁶⁹⁰ - LF multiple parcels
- PR #2618²⁶⁹¹ - Some fixes to libfabric that didn't get caught before the merge
- PR #2617²⁶⁹² - Adding `hpx::local_new`
- PR #2616²⁶⁹³ - Documentation: Extract all entities in order to autolink functions correctly
- Issue #2615²⁶⁹⁴ - Documentation: Linking functions is broken
- PR #2614²⁶⁹⁵ - Adding serialization for `std::deque`
- PR #2613²⁶⁹⁶ - We need to link with `boost.thread` and `boost.chrono` if we use `boost.context`
- PR #2612²⁶⁹⁷ - Making sure `for_loop_n(par, ...)` is actually executed in parallel
- PR #2611²⁶⁹⁸ - Add documentation to `invoke_fused` and friends NFC
- PR #2610²⁶⁹⁹ - Added reduction templates using an identity value
- PR #2608²⁷⁰⁰ - Fixing some unused vars in inspect
- PR #2607²⁷⁰¹ - Fixed build for mingw
- PR #2606²⁷⁰² - Supporting generic context for boost >= 1.61
- PR #2605²⁷⁰³ - Parcelport libfabric3
- PR #2604²⁷⁰⁴ - Adding allocator support to promise and friends
- PR #2603²⁷⁰⁵ - Barrier hang
- PR #2602²⁷⁰⁶ - Changes to scheduler to steal from one high-priority queue
- Issue #2601²⁷⁰⁷ - High priority tasks are not executed first
- PR #2600²⁷⁰⁸ - Compat fixes
- PR #2599²⁷⁰⁹ - Compatibility layer for threading support
- PR #2598²⁷¹⁰ - V1.1

2688 <https://github.com/STELLAR-GROUP/hpx/issues/2621>

2689 <https://github.com/STELLAR-GROUP/hpx/pull/2620>

2690 <https://github.com/STELLAR-GROUP/hpx/pull/2619>

2691 <https://github.com/STELLAR-GROUP/hpx/pull/2618>

2692 <https://github.com/STELLAR-GROUP/hpx/pull/2617>

2693 <https://github.com/STELLAR-GROUP/hpx/pull/2616>

2694 <https://github.com/STELLAR-GROUP/hpx/issues/2615>

2695 <https://github.com/STELLAR-GROUP/hpx/pull/2614>

2696 <https://github.com/STELLAR-GROUP/hpx/pull/2613>

2697 <https://github.com/STELLAR-GROUP/hpx/pull/2612>

2698 <https://github.com/STELLAR-GROUP/hpx/pull/2611>

2699 <https://github.com/STELLAR-GROUP/hpx/pull/2610>

2700 <https://github.com/STELLAR-GROUP/hpx/pull/2608>

2701 <https://github.com/STELLAR-GROUP/hpx/pull/2607>

2702 <https://github.com/STELLAR-GROUP/hpx/pull/2606>

2703 <https://github.com/STELLAR-GROUP/hpx/pull/2605>

2704 <https://github.com/STELLAR-GROUP/hpx/pull/2604>

2705 <https://github.com/STELLAR-GROUP/hpx/pull/2603>

2706 <https://github.com/STELLAR-GROUP/hpx/pull/2602>

2707 <https://github.com/STELLAR-GROUP/hpx/issues/2601>

2708 <https://github.com/STELLAR-GROUP/hpx/pull/2600>

2709 <https://github.com/STELLAR-GROUP/hpx/pull/2599>

2710 <https://github.com/STELLAR-GROUP/hpx/pull/2598>

- PR #2597²⁷¹¹ - Release V1.0
- PR #2592²⁷¹² - First attempt to introduce spmd_block in hpx
- PR #2586²⁷¹³ - local_segment in segmented_iterator_traits
- Issue #2584²⁷¹⁴ - Add allocator support to promise, packaged_task and friends
- PR #2576²⁷¹⁵ - Add missing dependencies of cuda based tests
- PR #2575²⁷¹⁶ - Remove warnings due to some captured variables
- Issue #2574²⁷¹⁷ - MSVC 2015 Compiler crash when building HPX
- Issue #2568²⁷¹⁸ - Remove throttle_scheduler as it has been abandoned
- Issue #2566²⁷¹⁹ - Add an inline versioning namespace before 1.0 release
- Issue #2565²⁷²⁰ - Raise minimal cmake version requirement
- PR #2556²⁷²¹ - Fixing scan partitioner
- PR #2546²⁷²² - Broadcast async
- Issue #2543²⁷²³ - make install fails due to a non-existing .so file
- PR #2495²⁷²⁴ - wait_or_add_new returning thread_id_type
- Issue #2480²⁷²⁵ - Unable to register new performance counter
- Issue #2471²⁷²⁶ - no type named ‘fcontext_t’ in namespace
- Issue #2456²⁷²⁷ - Re-implement hpx::util::unwrapped
- Issue #2455²⁷²⁸ - Add more arithmetic performance counters
- PR #2454²⁷²⁹ - Fix a couple of warnings and compiler errors
- PR #2453²⁷³⁰ - Timed executor support
- PR #2447²⁷³¹ - Implementing new executor API (P0443)
- Issue #2439²⁷³² - Implement executor proposal
- Issue #2408²⁷³³ - Stackoverflow detection for linux, e.g. based on libsigsev

²⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2597>

²⁷¹² <https://github.com/STELLAR-GROUP/hpx/pull/2592>

²⁷¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2586>

²⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2584>

²⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2576>

²⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2575>

²⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2574>

²⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2568>

²⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2566>

²⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2565>

²⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2556>

²⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/2546>

²⁷²³ <https://github.com/STELLAR-GROUP/hpx/issues/2543>

²⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2495>

²⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2480>

²⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2471>

²⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2456>

²⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2455>

²⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2454>

²⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2453>

²⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2447>

²⁷³² <https://github.com/STELLAR-GROUP/hpx/issues/2439>

²⁷³³ <https://github.com/STELLAR-GROUP/hpx/issues/2408>

- PR #2377²⁷³⁴ - Add a customization point for put_parcel so we can override actions
- Issue #2368²⁷³⁵ - HPX_ASSERT problem
- Issue #2324²⁷³⁶ - Change default number of threads used to the maximum of the system
- Issue #2266²⁷³⁷ - hpx_0.9.99 make tests fail
- PR #2195²⁷³⁸ - Support for code completion in VIM
- Issue #2137²⁷³⁹ - Hpx does not compile over osx
- Issue #2092²⁷⁴⁰ - make tests should just build the tests
- Issue #2026²⁷⁴¹ - Build HPX with Apple's clang
- Issue #1932²⁷⁴² - hpx with PBS fails on multiple localities
- PR #1914²⁷⁴³ - Parallel heap algorithm implementations WIP
- Issue #1598²⁷⁴⁴ - Disconnecting a locality results in segfault using heartbeat example
- Issue #1404²⁷⁴⁵ - unwrapped doesn't work with movable only types
- Issue #1400²⁷⁴⁶ - hpx::util::unwrapped doesn't work with non-future types
- Issue #1205²⁷⁴⁷ - TSS is broken
- Issue #1126²⁷⁴⁸ - vector<future<T>> does not work gracefully with dataflow, when_all and unwrapped
- Issue #1056²⁷⁴⁹ - Thread manager cleanup
- Issue #863²⁷⁵⁰ - Futures should not require a default constructor
- Issue #856²⁷⁵¹ - Allow runtimemode_connect to be used with security enabled
- Issue #726²⁷⁵² - Valgrind
- Issue #701²⁷⁵³ - Add RCR performance counter component
- Issue #528²⁷⁵⁴ - Add support for known failures and warning count/comparisons to hpx_run_tests.py

²⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2377>

²⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2368>

²⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2324>

²⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2266>

²⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2195>

²⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2137>

²⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2092>

²⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/2026>

²⁷⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1932>

²⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1914>

²⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1598>

²⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1404>

²⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1400>

²⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1205>

²⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1126>

²⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1056>

²⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/863>

²⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/856>

²⁷⁵² <https://github.com/STELLAR-GROUP/hpx/issues/726>

²⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/701>

²⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/528>

2.10.12 HPX V1.0.0 (Apr 24, 2017)

General changes

Here are some of the main highlights and changes for this release (in no particular order):

- Added the facility `hpx::split_future` which allows one to convert a `future<tuple<Ts...>>` into a `tuple<future<Ts>...>`. This functionality is not available when compiling *HPX* with VS2012.
- Added a new type of performance counter which allows one to return a list of values for each invocation. We also added a first counter of this type which collects a histogram of the times between parcels being created.
- Added new LCOs: `hpx::lcos::channel` and `hpx::lcos::local::channel` which are very similar to the well known channel constructs used in the Go language.
- Added new performance counters reporting the amount of data handled by the networking layer on a action-by-action basis (please see [PR #2289²⁷⁵⁵](#) 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 [libboost_simd](#). Please see [Issue #2333²⁷⁵⁶](#) 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-post²⁷⁵⁷](#) for more information.
- The parameter sequence for the `hpx::parallel::transform_reduce` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17. The old overload can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.
- The algorithm `hpx::parallel::inner_product` has been renamed to `hpx::parallel::transform_reduce` to match the changes this algorithm has undergone while being moved to C++17. The old `inner_product` names can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.
- Added versions of `hpx::get_ptr` taking client side representations for component instances as their parameter (instead of a global id).
- Added the helper utility `hpx::performance_counters::performance_counter_set` helping to encapsulate a set of performance counters to be managed concurrently.
- All execution policies and related classes have been renamed to be consistent with the naming changes applied for C++17. All policies now live in the namespace `hpx::parallel::execution`. The old names can be still enabled at configure time by specifying `-DHPX_WITH_EXECUTION_POLICY_COMPATIBILITY=On` to CMake.
- The thread scheduling subsystem has undergone a major refactoring which results in significant performance improvements. We have also improved the performance of creating `hpx::future` and of various facilities handling those.
- We have consolidated all of the code in `HPX.Compute` related to the integration of CUDA. `hpx::partitioned_vector` has been enabled to be usable with `hpx::compute::vector` which allows one to place the partitions on one or more GPU devices.
- Added new performance counters exposing various internals of the thread scheduling subsystem, such as the current idle- and busy-loop counters and instantaneous scheduler utilization.

²⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2289>

²⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2333>

²⁷⁵⁷ <http://stellar-group.org/2016/09/vectorized-cpp-parallel-algorithms-with-hpx/>

- 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 #2596²⁷⁵⁸ - Adding apex data
- PR #2595²⁷⁵⁹ - Remove obsolete file
- Issue #2594²⁷⁶⁰ - FindOpenCL.cmake mismatch with the official cmake module
- PR #2592²⁷⁶¹ - First attempt to introduce spmd_block in hpx
- Issue #2591²⁷⁶² - Feature request: continuation (then) which does not require the callable object to take a future<R> as parameter
- PR #2588²⁷⁶³ - Daint fixes
- PR #2587²⁷⁶⁴ - Fixing transfer_(continuation)_action::schedule
- PR #2585²⁷⁶⁵ - Work around MSVC having an ICE when compiling with -Ob2
- PR #2583²⁷⁶⁶ - changing 7zip command to 7za in roll_release.sh
- PR #2582²⁷⁶⁷ - First attempt to introduce spmd_block in hpx
- PR #2581²⁷⁶⁸ - Enable annotated function for parallel algorithms

²⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2596>

²⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2595>

²⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2594>

²⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2592>

²⁷⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2591>

²⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2588>

²⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2587>

²⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2585>

²⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2583>

²⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2582>

²⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2581>

- PR #2580²⁷⁶⁹ - First attempt to introduce spmd_block in hpx
- PR #2579²⁷⁷⁰ - Make thread NICE level setting an option
- PR #2578²⁷⁷¹ - Implementing enqueue instead of busy wait when no sender is available
- PR #2577²⁷⁷² - Retrieve -std=c++11 consistent nvcc flag
- PR #2576²⁷⁷³ - Add missing dependencies of cuda based tests
- PR #2575²⁷⁷⁴ - Remove warnings due to some captured variables
- PR #2573²⁷⁷⁵ - Attempt to resolve resolve_locality
- PR #2572²⁷⁷⁶ - Adding APEX hooks to background thread
- PR #2571²⁷⁷⁷ - Pick up hpx.ignore_batch_env from config map
- PR #2570²⁷⁷⁸ - Add commandline options –hpx:print-counters-locally
- PR #2569²⁷⁷⁹ - Fix computeapi unit tests
- PR #2567²⁷⁸⁰ - This adds another barrier::synchronize before registering performance counters
- PR #2564²⁷⁸¹ - Cray static toolchain support
- PR #2563²⁷⁸² - Fixed unhandled exception during startup
- PR #2562²⁷⁸³ - Remove partitioned_vector.cu from build tree when nvcc is used
- Issue #2561²⁷⁸⁴ - octo-tiger crash with commit 6e921495ff6c26f125d62629cbaad0525f14f7ab
- PR #2560²⁷⁸⁵ - Prevent -Wundef warnings on Vc version checks
- PR #2559²⁷⁸⁶ - Allowing CUDA callback to set the future directly from an OS thread
- PR #2558²⁷⁸⁷ - Remove warnings due to float precisions
- PR #2557²⁷⁸⁸ - Removing bogus handling of compile flags for CUDA
- PR #2556²⁷⁸⁹ - Fixing scan partitioner
- PR #2554²⁷⁹⁰ - Add more diagnostics to error thrown from find_appropriate_destination
- Issue #2555²⁷⁹¹ - No valid parcelport configured

²⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2580>

²⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2579>

²⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2578>

²⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2577>

²⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2576>

²⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2575>

²⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2573>

²⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2572>

²⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2571>

²⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2570>

²⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2569>

²⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2567>

²⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2564>

²⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2563>

²⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2562>

²⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2561>

²⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2560>

²⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2559>

²⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2558>

²⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2557>

²⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2556>

²⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2554>

²⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2555>

- PR #2553²⁷⁹² - Add cmake cuda_arch option
- PR #2552²⁷⁹³ - Remove incomplete datapar bindings to libflatarray
- PR #2551²⁷⁹⁴ - Rename hwloc_topology to hwloc_topology_info
- PR #2550²⁷⁹⁵ - Apex api updates
- PR #2549²⁷⁹⁶ - Pre-include defines.hpp to get the macro HPX_HAVE_CUDA value
- PR #2548²⁷⁹⁷ - Fixing issue with disconnect
- PR #2546²⁷⁹⁸ - Some fixes around cuda clang partitioned_vector example
- PR #2545²⁷⁹⁹ - Fix uses of the Vc2 datapar flags; the value, not the type, should be passed to functions
- PR #2542²⁸⁰⁰ - Make HPX_WITH_MALLOC easier to use
- PR #2541²⁸⁰¹ - avoid recompiles when enabling/disabling examples
- PR #2540²⁸⁰² - Fixing usage of target_link_libraries()
- PR #2539²⁸⁰³ - fix RPATH behaviour
- Issue #2538²⁸⁰⁴ - HPX_WITH_CUDA corrupts compilation flags
- PR #2537²⁸⁰⁵ - Add output of a Bazel Skylark extension for paths and compile options
- PR #2536²⁸⁰⁶ - Add counter exposing total available memory to Windows as well
- PR #2535²⁸⁰⁷ - Remove obsolete support for security
- Issue #2534²⁸⁰⁸ - Remove command line option --hpx:run-agas-server
- PR #2533²⁸⁰⁹ - Pre-cache locality endpoints during bootstrap
- PR #2532²⁸¹⁰ - Fixing handling of GIDs during serialization preprocessing
- PR #2531²⁸¹¹ - Amend uses of the term “functor”
- PR #2529²⁸¹² - added counter for reading available memory
- PR #2527²⁸¹³ - Facilities to create actions from lambdas
- PR #2526²⁸¹⁴ - Updated docs: HPX_WITH_EXAMPLES

²⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2553>

²⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2552>

²⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2551>

²⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2550>

²⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2549>

²⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2548>

²⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2546>

²⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2545>

²⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2542>

²⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2541>

²⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2540>

²⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2539>

²⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2538>

²⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2537>

²⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2536>

²⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2535>

²⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2534>

²⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2533>

²⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2532>

²⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2531>

²⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/2529>

²⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2527>

²⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2526>

- PR #2525²⁸¹⁵ - Remove warnings related to unused captured variables
- Issue #2524²⁸¹⁶ - CMAKE failed because it is missing: TCMALLOC_LIBRARY TCMALLOC_INCLUDE_DIR
- PR #2523²⁸¹⁷ - Fixing compose_cb stack overflow
- PR #2522²⁸¹⁸ - Instead of unlocking, ignore the lock while creating the message handler
- PR #2521²⁸¹⁹ - Create LPROGRESS_ logging macro to simplify progress tracking and timings
- PR #2520²⁸²⁰ - Intel 17 support
- PR #2519²⁸²¹ - Fix components example
- PR #2518²⁸²² - Fixing parcel scheduling
- Issue #2517²⁸²³ - Race condition during Parcel Coalescing Handler creation
- Issue #2516²⁸²⁴ - HPX locks up when using at least 256 localities
- Issue #2515²⁸²⁵ - error: Install cannot find “/lib/hpx/libparcel_coalescing.so.0.9.99” but I can see that file
- PR #2514²⁸²⁶ - Making sure that all continuations of a shared_future are invoked in order
- PR #2513²⁸²⁷ - Fixing locks held during suspension
- PR #2512²⁸²⁸ - MPI Parcelport improvements and fixes related to the background work changes
- PR #2511²⁸²⁹ - Fixing bit-wise (zero-copy) serialization
- Issue #2509²⁸³⁰ - Linking errors in hwloc_topology
- PR #2508²⁸³¹ - Added documentation for debugging with core files
- PR #2506²⁸³² - Fixing background work invocations
- PR #2505²⁸³³ - Fix tuple serialization
- Issue #2504²⁸³⁴ - Ensure continuations are called in the order they have been attached
- PR #2503²⁸³⁵ - Adding serialization support for Vc v2 (datapar)
- PR #2502²⁸³⁶ - Resolve various, minor compiler warnings
- PR #2501²⁸³⁷ - Some other fixes around cuda examples

²⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2525>

²⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2524>

²⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2523>

²⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2522>

²⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2521>

²⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2520>

²⁸²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2519>

²⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/2518>

²⁸²³ <https://github.com/STELLAR-GROUP/hpx/issues/2517>

²⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2516>

²⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2515>

²⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2514>

²⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2513>

²⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2512>

²⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2511>

²⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2509>

²⁸³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2508>

²⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/2506>

²⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/2505>

²⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2504>

²⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2503>

²⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2502>

²⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2501>

- Issue #2500²⁸³⁸ - nvcc / cuda clang issue due to a missing -DHPX_WITH_CUDA flag
- PR #2499²⁸³⁹ - Adding support for std::array to wait_all and friends
- PR #2498²⁸⁴⁰ - Execute background work as HPX thread
- PR #2497²⁸⁴¹ - Fixing configuration options for spinlock-deadlock detection
- PR #2496²⁸⁴² - Accounting for different compilers in CrayKNL toolchain file
- PR #2494²⁸⁴³ - Adding component base class which ties a component instance to a given executor
- PR #2493²⁸⁴⁴ - Enable controlling amount of pending threads which must be available to allow thread stealing
- PR #2492²⁸⁴⁵ - Adding new command line option –hpx:print-counter-reset
- PR #2491²⁸⁴⁶ - Resolve ambiguities when compiling with APEX
- PR #2490²⁸⁴⁷ - Resuming threads waiting on future with higher priority
- Issue #2489²⁸⁴⁸ - nvcc issue because -std=c++11 appears twice
- PR #2488²⁸⁴⁹ - Adding performance counters exposing the internal idle and busy-loop counters
- PR #2487²⁸⁵⁰ - Allowing for plain suspend to reschedule thread right away
- PR #2486²⁸⁵¹ - Only flag HPX code for CUDA if HPX_WITH_CUDA is set
- PR #2485²⁸⁵² - Making thread-queue parameters runtime-configurable
- PR #2484²⁸⁵³ - Added atomic counter for parcel-destinations
- PR #2483²⁸⁵⁴ - Added priority-queue lifo scheduler
- PR #2482²⁸⁵⁵ - Changing scheduler to steal only if more than a minimal number of tasks are available
- PR #2481²⁸⁵⁶ - Extending command line option –hpx:print-counter-destination to support value ‘none’
- PR #2479²⁸⁵⁷ - Added option to disable signal handler
- PR #2478²⁸⁵⁸ - Making sure the sine performance counter module gets loaded only for the corresponding example
- Issue #2477²⁸⁵⁹ - Breaking at a throw statement
- PR #2476²⁸⁶⁰ - Annotated function

²⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2500>

²⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2499>

²⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2498>

²⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2497>

²⁸⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2496>

²⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2494>

²⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2493>

²⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2492>

²⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2491>

²⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2490>

²⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2489>

²⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2488>

²⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2487>

²⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2486>

²⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2485>

²⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2484>

²⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2483>

²⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2482>

²⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2481>

²⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2479>

²⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2478>

²⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2477>

²⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2476>

- PR #2475²⁸⁶¹ - Ensure that using %osthread% during logging will not throw for non-hpx threads
- PR #2474²⁸⁶² - Remove now superficial non_direct actions from base_lco and friends
- PR #2473²⁸⁶³ - Refining support for ITTNotify
- PR #2472²⁸⁶⁴ - Some fixes around hpx compute
- Issue #2470²⁸⁶⁵ - redefinition of boost::detail::spinlock
- Issue #2469²⁸⁶⁶ - Dataflow performance issue
- PR #2468²⁸⁶⁷ - Perf docs update
- PR #2466²⁸⁶⁸ - Guarantee to execute remote direct actions on HPX-thread
- PR #2465²⁸⁶⁹ - Improve demo : Async copy and fixed device handling
- PR #2464²⁸⁷⁰ - Adding performance counter exposing instantaneous scheduler utilization
- PR #2463²⁸⁷¹ - Downcast to future<void>
- PR #2462²⁸⁷² - Fixed usage of ITT-Notify API with Intel Amplifier
- PR #2461²⁸⁷³ - Cublas demo
- PR #2460²⁸⁷⁴ - Fixing thread bindings
- PR #2459²⁸⁷⁵ - Make -std=c++11 nvcc flag consistent for in-build and installed versions
- Issue #2457²⁸⁷⁶ - Segmentation fault when registering a partitioned vector
- PR #2452²⁸⁷⁷ - Properly releasing global barrier for unhandled exceptions
- PR #2451²⁸⁷⁸ - Fixing long shutdown times
- PR #2450²⁸⁷⁹ - Attempting to fix initialization errors on newer platforms (Boost V1.63)
- PR #2449²⁸⁸⁰ - Replace BOOST_COMPILER_FENCE with an HPX version
- PR #2448²⁸⁸¹ - This fixes a possible race in the migration code
- PR #2445²⁸⁸² - **Fixing dataflow et.al. for futures or future-ranges wrapped into ref()**
- PR #2444²⁸⁸³ - Fix segfaults

²⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2475>

²⁸⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2474>

²⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2473>

²⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2472>

²⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2470>

²⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2469>

²⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2468>

²⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2466>

²⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2465>

²⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2464>

²⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2463>

²⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2462>

²⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2461>

²⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2460>

²⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2459>

²⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2457>

²⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2452>

²⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2451>

²⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2450>

²⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2449>

²⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2448>

²⁸⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2445>

²⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2444>

- PR #2443²⁸⁸⁴ - Issue 2442
- Issue #2442²⁸⁸⁵ - Mismatch between #if/#endif and namespace scope brackets in this_thread_executors.hpp
- Issue #2441²⁸⁸⁶ - undeclared identifier BOOST_COMPILER_FENCE
- PR #2440²⁸⁸⁷ - Knl build
- PR #2438²⁸⁸⁸ - Datapar backend
- PR #2437²⁸⁸⁹ - Adapt algorithm parameter sequence changes from C++17
- PR #2436²⁸⁹⁰ - Adapt execution policy name changes from C++17
- Issue #2435²⁸⁹¹ - Trunk broken, undefined reference to hpx::thread::interrupt(hpx::thread::id, bool)
- PR #2434²⁸⁹² - More fixes to resource manager
- PR #2433²⁸⁹³ - Added versions of hpx::get_ptr taking client side representations
- PR #2432²⁸⁹⁴ - Warning fixes
- PR #2431²⁸⁹⁵ - Adding facility representing set of performance counters
- PR #2430²⁸⁹⁶ - Fix parallel_executor thread spawning
- PR #2429²⁸⁹⁷ - Fix attribute warning for gcc
- Issue #2427²⁸⁹⁸ - Seg fault running octo-tiger with latest HPX commit
- Issue #2426²⁸⁹⁹ - Bug in 9592f5c0bc29806fce0dbe73f35b6ca7e027edcb causes immediate crash in Octo-tiger
- PR #2425²⁹⁰⁰ - Fix nvcc errors due to constexpr specifier
- Issue #2424²⁹⁰¹ - Async action on component present on hpx::find_here is executing synchronously
- PR #2423²⁹⁰² - Fix nvcc errors due to constexpr specifier
- PR #2422²⁹⁰³ - Implementing hpx::this_thread thread data functions
- PR #2421²⁹⁰⁴ - Adding benchmark for wait_all
- Issue #2420²⁹⁰⁵ - Returning object of a component client from another component action fails
- PR #2419²⁹⁰⁶ - Infiniband parcelport

²⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2443>

²⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2442>

²⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2441>

²⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2440>

²⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2438>

²⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2437>

²⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2436>

²⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2435>

²⁸⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2434>

²⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2433>

²⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2432>

²⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2431>

²⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2430>

²⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2429>

²⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2427>

²⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2426>

²⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2425>

²⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/2424>

²⁹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2423>

²⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2422>

²⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2421>

²⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2420>

²⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2419>

- Issue #2418²⁹⁰⁷ - gcc + nvcc fails to compile code that uses partitioned_vector
- PR #2417²⁹⁰⁸ - Fixing context switching
- PR #2416²⁹⁰⁹ - Adding fixes and workarounds to allow compilation with nvcc/msvc (VS2015up3)
- PR #2415²⁹¹⁰ - Fix errors coming from hpx compute examples
- PR #2414²⁹¹¹ - Fixing msvc12
- PR #2413²⁹¹² - Enable cuda/nvcc or cuda/clang when using add_hpx_executable()
- PR #2412²⁹¹³ - Fix issue in HPX_SetupTarget.cmake when cuda is used
- PR #2411²⁹¹⁴ - This fixes the core compilation issues with MSVC12
- Issue #2410²⁹¹⁵ - undefined reference to opal_hwloc191_hwloc_.....
- PR #2409²⁹¹⁶ - Fixing locking for channel and receive_buffer
- PR #2407²⁹¹⁷ - Solving #2402 and #2403
- PR #2406²⁹¹⁸ - Improve guards
- PR #2405²⁹¹⁹ - Enable parallel::for_each for iterators returning proxy types
- PR #2404²⁹²⁰ - Forward the explicitly given result_type in the hpx invoke
- Issue #2403²⁹²¹ - datapar_execution + zip iterator: lambda arguments aren't references
- Issue #2402²⁹²² - datapar algorithm instantiated with wrong type #2402
- PR #2401²⁹²³ - Added support for imported libraries to HPX_Libraries.cmake
- PR #2400²⁹²⁴ - Use CMake policy CMP0060
- Issue #2399²⁹²⁵ - Error trying to push back vector of futures to vector
- PR #2398²⁹²⁶ - Allow config #defines to be written out to custom config/defines.hpp
- Issue #2397²⁹²⁷ - CMake generated config defines can cause tedious rebuilds category
- Issue #2396²⁹²⁸ - BOOST_ROOT paths are not used at link time
- PR #2395²⁹²⁹ - Fix target_link_libraries() issue when HPX Cuda is enabled

²⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2418>

²⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2417>

²⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2416>

²⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2415>

²⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2414>

²⁹¹² <https://github.com/STELLAR-GROUP/hpx/pull/2413>

²⁹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2412>

²⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2411>

²⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2410>

²⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2409>

²⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2407>

²⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2406>

²⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2405>

²⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2404>

²⁹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/2403>

²⁹²² <https://github.com/STELLAR-GROUP/hpx/issues/2402>

²⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/2401>

²⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2400>

²⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2399>

²⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2398>

²⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2397>

²⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2396>

²⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2395>

- Issue #2394²⁹³⁰ - Template compilation error using HPX_WITH_DATAPAR_LIBFLATARRAY
- PR #2393²⁹³¹ - Fixing lock registration for recursive mutex
- PR #2392²⁹³² - Add keywords in target_link_libraries in hpx_setup_target
- PR #2391²⁹³³ - Clang goroutines
- Issue #2390²⁹³⁴ - Adapt execution policy name changes from C++17
- PR #2389²⁹³⁵ - Chunk allocator and pool are not used and are obsolete
- PR #2388²⁹³⁶ - Adding functionalities to datapar needed by octotiger
- PR #2387²⁹³⁷ - Fixing race condition for early parcels
- Issue #2386²⁹³⁸ - Lock registration broken for recursive_mutex
- PR #2385²⁹³⁹ - Datapar zip iterator
- PR #2384²⁹⁴⁰ - Fixing race condition in for_loop_reduction
- PR #2383²⁹⁴¹ - Continuations
- PR #2382²⁹⁴² - add LibFlatArray-based backend for datapar
- PR #2381²⁹⁴³ - remove unused typedef to get rid of compiler warnings
- PR #2380²⁹⁴⁴ - Tau cleanup
- PR #2379²⁹⁴⁵ - Can send immediate
- PR #2378²⁹⁴⁶ - Renaming copy_helper/copy_n_helper/move_helper/move_n_helper
- Issue #2376²⁹⁴⁷ - Boost trunk's spinlock initializer fails to compile
- PR #2375²⁹⁴⁸ - Add support for minimal thread local data
- PR #2374²⁹⁴⁹ - Adding API functions set_config_entry_callback
- PR #2373²⁹⁵⁰ - Add a simple utility for debugging that gives suspended task backtraces
- PR #2372²⁹⁵¹ - Barrier Fixes
- Issue #2370²⁹⁵² - Can't wait on a wrapped future

²⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2394>

²⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2393>

²⁹³² <https://github.com/STELLAR-GROUP/hpx/pull/2392>

²⁹³³ <https://github.com/STELLAR-GROUP/hpx/pull/2391>

²⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2390>

²⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2389>

²⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2388>

²⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2387>

²⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2386>

²⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2385>

²⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2384>

²⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2383>

²⁹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2382>

²⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2381>

²⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2380>

²⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2379>

²⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2378>

²⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2376>

²⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2375>

²⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2374>

²⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2373>

²⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2372>

²⁹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2370>

- PR #2369²⁹⁵³ - Fixing stable_partition
- PR #2367²⁹⁵⁴ - Fixing find_prefixes for Windows platforms
- PR #2366²⁹⁵⁵ - Testing for experimental/optional only in C++14 mode
- PR #2364²⁹⁵⁶ - Adding set_config_entry
- PR #2363²⁹⁵⁷ - Fix papi
- PR #2362²⁹⁵⁸ - Adding missing macros for new non-direct actions
- PR #2361²⁹⁵⁹ - Improve cmake output to help debug compiler incompatibility check
- PR #2360²⁹⁶⁰ - Fixing race condition in condition_variable
- PR #2359²⁹⁶¹ - Fixing shutdown when parcels are still in flight
- Issue #2357²⁹⁶² - failed to insert console_print_action into typename_to_id_t registry
- PR #2356²⁹⁶³ - Fixing return type of get_iterator_tuple
- PR #2355²⁹⁶⁴ - Fixing compilation against Boost 1.62
- PR #2354²⁹⁶⁵ - Adding serialization for mask_type if CPU_COUNT > 64
- PR #2353²⁹⁶⁶ - Adding hooks to tie in APEX into the parcel layer
- Issue #2352²⁹⁶⁷ - Compile errors when using intel 17 beta (for KNL) on edison
- PR #2351²⁹⁶⁸ - Fix function vtable get_function_address implementation
- Issue #2350²⁹⁶⁹ - Build failure - master branch (4de09f5) with Intel Compiler v17
- PR #2349²⁹⁷⁰ - Enabling zero-copy serialization support for std::vector<>
- PR #2348²⁹⁷¹ - Adding test to verify #2334 is fixed
- PR #2347²⁹⁷² - Bug fixes for hpx.compute and hpx::lcos::channel
- PR #2346²⁹⁷³ - Removing cmake “find” files that are in the APEX cmake Modules
- PR #2345²⁹⁷⁴ - Implemented parallel::stable_partition
- PR #2344²⁹⁷⁵ - Making hpx::lcos::channel usable with basename registration

²⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2369>

²⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2367>

²⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2366>

²⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2364>

²⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2363>

²⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2362>

²⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2361>

²⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2360>

²⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2359>

²⁹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2357>

2963 <https://github.com/STELLAR-GROUP/hpx/pull/2356>2964 <https://github.com/STELLAR-GROUP/hpx/pull/2355>2965 <https://github.com/STELLAR-GROUP/hpx/pull/2354>2966 <https://github.com/STELLAR-GROUP/hpx/pull/2353>2967 <https://github.com/STELLAR-GROUP/hpx/issues/2352>2968 <https://github.com/STELLAR-GROUP/hpx/pull/2351>2969 <https://github.com/STELLAR-GROUP/hpx/issues/2350>2970 <https://github.com/STELLAR-GROUP/hpx/pull/2349>2971 <https://github.com/STELLAR-GROUP/hpx/pull/2348>2972 <https://github.com/STELLAR-GROUP/hpx/pull/2347>2973 <https://github.com/STELLAR-GROUP/hpx/pull/2346>2974 <https://github.com/STELLAR-GROUP/hpx/pull/2345>2975 <https://github.com/STELLAR-GROUP/hpx/pull/2344>

- PR #2343²⁹⁷⁶ - Fix a couple of examples that failed to compile after recent api changes
- Issue #2342²⁹⁷⁷ - Enabling APEX causes link errors
- PR #2341²⁹⁷⁸ - Removing cmake “find” files that are in the APEX cmake Modules
- PR #2340²⁹⁷⁹ - Implemented all existing datapar algorithms using Boost.SIMD
- PR #2339²⁹⁸⁰ - Fixing 2338
- PR #2338²⁹⁸¹ - Possible race in sliding semaphore
- PR #2337²⁹⁸² - Adjust osu_latency test to measure window_size parcels in flight at once
- PR #2336²⁹⁸³ - Allowing remote direct actions to be executed without spawning a task
- PR #2335²⁹⁸⁴ - Making sure multiple components are properly initialized from arguments
- Issue #2334²⁹⁸⁵ - Cannot construct component with large vector on a remote locality
- PR #2332²⁹⁸⁶ - Fixing hpx::lcos::local::barrier
- PR #2331²⁹⁸⁷ - Updating APEX support to include OTF2
- PR #2330²⁹⁸⁸ - Support for data-parallelism for parallel algorithms
- Issue #2329²⁹⁸⁹ - Coordinate settings in cmake
- PR #2328²⁹⁹⁰ - fix LibGeoDecomp builds with HPX + GCC 5.3.0 + CUDA 8RC
- PR #2326²⁹⁹¹ - Making scan_partitioner work (for now)
- Issue #2323²⁹⁹² - Constructing a vector of components only correctly initializes the first component
- PR #2322²⁹⁹³ - Fix problems that bubbled up after merging #2278
- PR #2321²⁹⁹⁴ - Scalable barrier
- PR #2320²⁹⁹⁵ - Std flag fixes
- Issue #2319²⁹⁹⁶ - -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 #2318²⁹⁹⁷ - Improve handling of -hpx:bind=<bind-spec>
- PR #2317²⁹⁹⁸ - Making sure command line warnings are printed once only

²⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2343>

²⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2342>

²⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2341>

²⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2340>

²⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2339>

²⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2338>

²⁹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2337>

²⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2336>

²⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2335>

²⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2334>

²⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2332>

²⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2331>

²⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2330>

²⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2329>

²⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2328>

²⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2326>

²⁹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/2323>

²⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2322>

²⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2321>

²⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2320>

²⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2319>

²⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2318>

²⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2317>

- PR #2316²⁹⁹⁹ - Fixing command line handling for default bind mode
- PR #2315³⁰⁰⁰ - Set id_retrieved if set_id is present
- Issue #2314³⁰⁰¹ - Warning for requested/allocated thread discrepancy is printed twice
- Issue #2313³⁰⁰² - --hpx:print-bind doesn't work with --hpx:pu-step
- Issue #2312³⁰⁰³ - --hpx:bind range specifier restrictions are overly restrictive
- Issue #2311³⁰⁰⁴ - hpx_0.9.99 out of project build fails
- PR #2310³⁰⁰⁵ - Simplify function registration
- PR #2309³⁰⁰⁶ - Spelling and grammar revisions in documentation (and some code)
- PR #2306³⁰⁰⁷ - Correct minor typo in the documentation
- PR #2305³⁰⁰⁸ - Cleaning up and fixing parcel coalescing
- PR #2304³⁰⁰⁹ - Inspect checks for stream related includes
- PR #2303³⁰¹⁰ - Add functionality allowing to enumerate threads of given state
- PR #2301³⁰¹¹ - Algorithm overloads fix for VS2013
- PR #2300³⁰¹² - Use <cstdint>, add inspect checks
- PR #2299³⁰¹³ - Replace boost::[c]ref with std::[c]ref, add inspect checks
- PR #2297³⁰¹⁴ - Fixing compilation with no hw_loc
- PR #2296³⁰¹⁵ - Hp compute
- PR #2295³⁰¹⁶ - Making sure for_loop(execution::par, 0, N, ...) is actually executed in parallel
- PR #2294³⁰¹⁷ - Throwing exceptions if the runtime is not up and running
- PR #2293³⁰¹⁸ - Removing unused parcel port code
- PR #2292³⁰¹⁹ - Refactor function vtables
- PR #2291³⁰²⁰ - Fixing 2286
- PR #2290³⁰²¹ - Simplify algorithm overloads

²⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2316>

³⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2315>

³⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/2314>

³⁰⁰² <https://github.com/STELLAR-GROUP/hpx/issues/2313>

³⁰⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2312>

³⁰⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2311>

³⁰⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2310>

³⁰⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2309>

³⁰⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2306>

3008 <https://github.com/STELLAR-GROUP/hpx/pull/2305>3009 <https://github.com/STELLAR-GROUP/hpx/pull/2304>3010 <https://github.com/STELLAR-GROUP/hpx/pull/2303>

3011 <https://github.com/STELLAR-GROUP/hpx/pull/2301>

3012 <https://github.com/STELLAR-GROUP/hpx/pull/2300>

3013 <https://github.com/STELLAR-GROUP/hpx/pull/2299>

3014 <https://github.com/STELLAR-GROUP/hpx/pull/2297>

3015 <https://github.com/STELLAR-GROUP/hpx/pull/2296>

3016 <https://github.com/STELLAR-GROUP/hpx/pull/2295>

3017 <https://github.com/STELLAR-GROUP/hpx/pull/2294>

3018 <https://github.com/STELLAR-GROUP/hpx/pull/2293>

3019 <https://github.com/STELLAR-GROUP/hpx/pull/2292>

3020 <https://github.com/STELLAR-GROUP/hpx/pull/2291>

3021 <https://github.com/STELLAR-GROUP/hpx/pull/2290>

2.10. Releases

1781

- PR #2289³⁰²² - Adding performance counters reporting parcel related data on a per-action basis
- Issue #2288³⁰²³ - Remove dormant parcelports
- Issue #2286³⁰²⁴ - adjustments to parcel handling to support parcelports that do not need a connection cache
- PR #2285³⁰²⁵ - add CMake option to disable package export
- PR #2283³⁰²⁶ - Add more inspect checks for use of deprecated components
- Issue #2282³⁰²⁷ - Arithmetic exception in executor static chunker
- Issue #2281³⁰²⁸ - For loop doesn't parallelize
- PR #2280³⁰²⁹ - Fixing 2277: build failure with PAPI
- PR #2279³⁰³⁰ - Child vs parent stealing
- Issue #2277³⁰³¹ - master branch build failure (53c5b4f) with papi
- PR #2276³⁰³² - Compile time launch policies
- PR #2275³⁰³³ - Replace boost::chrono with std::chrono in interfaces
- PR #2274³⁰³⁴ - Replace most uses of Boost.Assign with initializer list
- PR #2273³⁰³⁵ - Fixed typos
- PR #2272³⁰³⁶ - Inspect checks
- PR #2270³⁰³⁷ - Adding test verifying -Ihpx.os_threads=all
- PR #2269³⁰³⁸ - Added inspect check for now obsolete boost type traits
- PR #2268³⁰³⁹ - Moving more code into source files
- Issue #2267³⁰⁴⁰ - Add inspect support to deprecate Boost.TypeTraits
- PR #2265³⁰⁴¹ - Adding channel LCO
- PR #2264³⁰⁴² - Make support for std::ref mandatory
- PR #2263³⁰⁴³ - Constrain tuple_member forwarding constructor
- Issue #2262³⁰⁴⁴ - Test hpx.os_threads=all

³⁰²² <https://github.com/STELLAR-GROUP/hpx/pull/2289>

³⁰²³ <https://github.com/STELLAR-GROUP/hpx/issues/2288>

³⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/issu.../issues/2286>

³⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2285>

³⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2283>

³⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/issu.../issues/2282>

³⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/issu.../issues/2281>

³⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2280>

³⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2279>

³⁰³¹ <https://github.com/STELLAR-GROUP/hpx/issu.../issues/2277>

³⁰³² <https://github.com/STELLAR-GROUP/hpx/pull/2276>

³⁰³³ <https://github.com/STELLAR-GROUP/hpx/pull/2275>

³⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2274>

³⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2273>

³⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2272>

³⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2270>

³⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2269>

³⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2268>

³⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issu.../issues/2267>

³⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2265>

³⁰⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2264>

³⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2263>

³⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issu.../issues/2262>

- Issue #2261³⁰⁴⁵ - OS X: Error: no matching constructor for initialization of ‘hpx::lcos::local::condition_variable_any’
- Issue #2260³⁰⁴⁶ - Make support for std::ref mandatory
- PR #2259³⁰⁴⁷ - Remove most of Boost.MPL, Boost.EnableIf and Boost.TypeTraits
- PR #2258³⁰⁴⁸ - Fixing #2256
- PR #2257³⁰⁴⁹ - Fixing launch process
- Issue #2256³⁰⁵⁰ - Actions are not registered if not invoked
- PR #2255³⁰⁵¹ - Coalescing histogram
- PR #2254³⁰⁵² - Silence explicit initialization in copy-constructor warnings
- PR #2253³⁰⁵³ - Drop support for GCC 4.6 and 4.7
- PR #2252³⁰⁵⁴ - Prepare V1.0
- PR #2251³⁰⁵⁵ - Convert to 0.9.99
- PR #2249³⁰⁵⁶ - Adding iterator_facade and iterator_adaptor
- Issue #2248³⁰⁵⁷ - Need a feature to yield to a new task immediately
- PR #2246³⁰⁵⁸ - Adding split_future
- PR #2245³⁰⁵⁹ - Add an example for handing over a component instance to a dynamically launched locality
- Issue #2243³⁰⁶⁰ - Add example demonstrating AGAS symbolic name registration
- Issue #2242³⁰⁶¹ - pkgconfig test broken on CentOS 7 / Boost 1.61
- Issue #2241³⁰⁶² - Compilation error for partitioned vector in hpx_compute branch
- PR #2240³⁰⁶³ - Fixing termination detection on one locality
- Issue #2239³⁰⁶⁴ - Create a new facility lcos::split_all
- Issue #2236³⁰⁶⁵ - hpx::cout vs. std::cout
- PR #2232³⁰⁶⁶ - Implement local-only primary namespace service
- Issue #2147³⁰⁶⁷ - would like to know how much data is being routed by particular actions

³⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2261>

³⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2260>

³⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2259>

³⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2258>

³⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2257>

³⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2256>

³⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2255>

³⁰⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2254>

³⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2253>

³⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2252>

³⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2251>

³⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2249>

³⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2248>

³⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2246>

³⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2245>

³⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2243>

³⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2242>

³⁰⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2241>

³⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2240>

³⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2239>

³⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2236>

³⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2232>

³⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2147>

- Issue #2109³⁰⁶⁸ - Warning while compiling hpx
- Issue #1973³⁰⁶⁹ - Setting INTERFACE_COMPILE_OPTIONS for hpx_init in CMake taints Fortran_FLAGS
- Issue #1864³⁰⁷⁰ - run_guarded using bound function ignores reference
- Issue #1754³⁰⁷¹ - Running with TCP parcelport causes immediate crash or freeze
- Issue #1655³⁰⁷² - Enable zip_iterator to be used with Boost traversal iterator categories
- Issue #1591³⁰⁷³ - Optimize AGAS for shared memory only operation
- Issue #1401³⁰⁷⁴ - Need an efficient infiniband parcelport
- Issue #1125³⁰⁷⁵ - Fix the IPC parcelport
- Issue #839³⁰⁷⁶ - Refactor ibverbs and shmem parcelport
- Issue #702³⁰⁷⁷ - Add instrumentation of parcel layer
- Issue #668³⁰⁷⁸ - Implement ispc task interface
- Issue #533³⁰⁷⁹ - Thread queue/deque internal parameters should be runtime configurable
- Issue #475³⁰⁸⁰ - Create a means of combining performance counters into querysets

2.10.13 HPX V0.9.99 (Jul 15, 2016)

General changes

As the version number of this release hints, we consider this release to be a preview for the upcoming *HPX* V1.0. All of the functionalities we set out to implement for V1.0 are in place; all of the features we wanted to have exposed are ready. We are very happy with the stability and performance of *HPX* and we would like to present this release to the community in order for us to gather broad feedback before releasing V1.0. We still expect for some minor details to change, but on the whole this release represents what we would like to have in a V1.0.

Overall, since the last release we have had almost 1600 commits while closing almost 400 tickets. These numbers reflect the incredible development activity we have seen over the last couple of months. We would like to express a big ‘Thank you!’ to all contributors and those who helped to make this release happen.

The most notable addition in terms of new functionality available with this release is the full implementation of object migration (i.e. the ability to transparently move *HPX* components to a different compute node). Additionally, this release of *HPX* cleans up many minor issues and some API inconsistencies.

Here are some of the main highlights and changes for this release (in no particular order):

- We have fixed a couple of issues in AGAS and the parcel layer which have caused hangs, segmentation faults at exit, and a slowdown of applications over time. Fixing those has significantly increased the overall stability and performance of distributed runs.

³⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2109>

³⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1973>

³⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1864>

³⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1754>

³⁰⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1655>

³⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1591>

³⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1401>

³⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1125>

³⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/839>

³⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/702>

³⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/668>

³⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/533>

³⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/475>

- We have started to add parallel algorithm overloads based on the C++ Extensions for Ranges ([N4560³⁰⁸¹](#)) proposal. This also includes the addition of projections to the existing algorithms. Please see [Issue #1668³⁰⁸²](#) for a list of algorithms which have been adapted to [N4560³⁰⁸³](#).
- We have implemented index-based parallel for-loops based on a corresponding standardization proposal ([P0075R1³⁰⁸⁴](#)). Please see [Issue #2016³⁰⁸⁵](#) for a list of available algorithms.
- We have added implementations for more parallel algorithms as proposed for the upcoming C++ 17 Standard. See [Issue #1141³⁰⁸⁶](#) 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 #559³⁰⁸⁷](#) and [PR #1966³⁰⁸⁸](#) 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 #1265³⁰⁸⁹](#) and [PR #1912³⁰⁹⁰](#) for more information.
- Adding overloads of `gather_here` and `gather_there` to accept the plain values of the data to gather (in addition to the existing overloads expecting futures).
- We have cleaned up and refactored large parts of the code base. This helped reducing compile and link times of *HPX* itself and also of applications depending on it. We have further decreased the dependency of *HPX* on the Boost libraries by replacing part of those with facilities available from the standard libraries.
- Wherever possible we have removed dependencies of our API on Boost by replacing those with the equivalent facility from the C++11 standard library.
- We have added new performance counters for parcel coalescing, file-IO, the AGAS cache, and overall scheduler time. Resetting performance counters has been overhauled and fixed.
- We have introduced a generic client type `hpx::components::client<>` and added support for using it with `hpx::async`. This removes the necessity to implement specific client types for every component type without losing type safety. This deemphasizes the need for using the low level `hpx::id_type` for referencing

³⁰⁸¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf>

³⁰⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1668>

³⁰⁸³ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf>

³⁰⁸⁴ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0075r1.pdf>

³⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2016>

³⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1141>

³⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/559>

³⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1966>

³⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1265>

³⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1912>

(possibly remote) component instances. The plan is to deprecate the direct use of `hpx::id_type` in user code in the future.

- We have added a special iterator which supports automatic prefetching of one or more arrays for speeding up loop-like code (see `hpx::parallel::util::make_prefetcher_context()`).
- We have extended the interfaces exposed from executors (as proposed by [N4406³⁰⁹¹](#)) 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 [CMake³⁰⁹²](#) 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 ([N4560³⁰⁹³](#)) proposal imposes some breaking changes related to the return types of some of the parallel algorithms. Please see [Issue #1668³⁰⁹⁴](#) for a list of algorithms which have already been adapted.
- The facility `hpx::lcos::make_future_void()` has been replaced by `hpx::make_future<void>()`.
- We have removed support for Intel V13 and gcc 4.4.x.
- We have removed (default) support for the generic `hpx::parallel::execution_policy` because it was removed from the Parallelism TS ([_cpp11_n4104_](#)) while it was being added to the upcoming C++17 Standard. This facility can be still enabled at configure time by specifying `-DHPX_WITH_GENERIC_EXECUTION_POLICY=On` to CMake.
- Uses of `boost::shared_ptr` and related facilities have been replaced with `std::shared_ptr` and friends. Uses of `boost::unique_lock`, `boost::lock_guard` etc. have also been replaced by the equivalent (and equally named) tools available from the C++11 standard library.
- Facilities that used to expect an explicit `boost::unique_lock` now take an `std::unique_lock`. Additionally, `condition_variable` no longer aliases `condition_variable_any`; its interface now only works with `std::unique_lock<local::mutex>`.
- Uses of `boost::function`, `boost::bind`, `boost::tuple` have been replaced by the corresponding facilities in *HPX* (`hpx::util::function`, `hpx::util::bind`, and `hpx::util::tuple`, respectively).

³⁰⁹¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf>

³⁰⁹² <https://www.cmake.org>

³⁰⁹³ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf>

³⁰⁹⁴ <https://github.com/STEllAR-GROUP/hpx/issues/1668>

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2250³⁰⁹⁵ - change default chunker of parallel executor to static one
- PR #2247³⁰⁹⁶ - HPX on ppc64le
- PR #2244³⁰⁹⁷ - Fixing MSVC problems
- PR #2238³⁰⁹⁸ - Fixing small typos
- PR #2237³⁰⁹⁹ - Fixing small typos
- PR #2234³¹⁰⁰ - Fix broken add test macro when extra args are passed in
- PR #2231³¹⁰¹ - Fixing possible race during future awaiting in serialization
- PR #2230³¹⁰² - Fix stream nvcc
- PR #2229³¹⁰³ - Fixed run_as_hpx_thread
- PR #2228³¹⁰⁴ - On prefetching_test branch : adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2227³¹⁰⁵ - Support for HPXCL's opencl::event
- PR #2226³¹⁰⁶ - Preparing for release of V0.9.99
- PR #2225³¹⁰⁷ - fix issue when compiling components with hpxcxx
- PR #2224³¹⁰⁸ - Compute alloc fix
- PR #2223³¹⁰⁹ - Simplify promise
- PR #2222³¹¹⁰ - Replace last uses of boost::function by util::function_nonsr
- PR #2221³¹¹¹ - Fix config tests
- PR #2220³¹¹² - Fixing gcc 4.6 compilation issues
- PR #2219³¹¹³ - nullptr support for [unique_]function
- PR #2218³¹¹⁴ - Introducing clang tidy
- PR #2216³¹¹⁵ - Replace NULL with nullptr

³⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2250>

³⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2247>

³⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2244>

³⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2238>

³⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2237>

³¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2234>

³¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2231>

³¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2230>

³¹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2229>

³¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2228>

³¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2227>

³¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2226>

³¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2225>

³¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2224>

³¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2223>

³¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2222>

³¹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2221>

³¹¹² <https://github.com/STELLAR-GROUP/hpx/pull/2220>

³¹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2219>

³¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2218>

³¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2216>

- Issue #2214³¹¹⁶ - Let inspect flag use of NULL, suggest nullptr instead
- PR #2213³¹¹⁷ - Require support for nullptr
- PR #2212³¹¹⁸ - Properly find jemalloc through pkg-config
- PR #2211³¹¹⁹ - Disable a couple of warnings reported by Intel on Windows
- PR #2210³¹²⁰ - Fixed host::block_allocator::bulk_construct
- PR #2209³¹²¹ - Started to clean up new sort algorithms, made things compile for sort_by_key
- PR #2208³¹²² - A couple of fixes that were exposed by a new sort algorithm
- PR #2207³¹²³ - Adding missing includes in /hpx/include/serialization.hpp
- PR #2206³¹²⁴ - Call package_action::get_future before package_action::apply
- PR #2205³¹²⁵ - The indirect_packaged_task::operator() needs to be run on a HPX thread
- PR #2204³¹²⁶ - Variadic executor parameters
- PR #2203³¹²⁷ - Delay-initialize members of partitioned iterator
- PR #2202³¹²⁸ - Added segmented fill for hpx::vector
- Issue #2201³¹²⁹ - Null Thread id encountered on partitioned_vector
- PR #2200³¹³⁰ - Fix hangs
- PR #2199³¹³¹ - Deprecating hpx/traits.hpp
- PR #2198³¹³² - Making explicit inclusion of external libraries into build
- PR #2197³¹³³ - Fix typo in QT CMakeLists
- PR #2196³¹³⁴ - Fixing a gcc warning about attributes being ignored
- PR #2194³¹³⁵ - Fixing partitioned_vector_spmd_FOREACH example
- Issue #2193³¹³⁶ - partitioned_vector_spmd_FOREACH seg faults
- PR #2192³¹³⁷ - Support Boost.Thread v4
- PR #2191³¹³⁸ - HPX.Compute prototype

³¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2214>

³¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2213>

³¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2212>

³¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2211>

³¹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2210>

³¹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2209>

³¹²² <https://github.com/STELLAR-GROUP/hpx/pull/2208>

³¹²³ <https://github.com/STELLAR-GROUP/hpx/pull/2207>

³¹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2206>

³¹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2205>

³¹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2204>

³¹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2203>

³¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2202>

³¹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2201>

³¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2200>

³¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2199>

³¹³² <https://github.com/STELLAR-GROUP/hpx/pull/2198>

³¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/2197>

³¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2196>

³¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2194>

³¹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2193>

³¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2192>

³¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2191>

- PR #2190³¹³⁹ - Spawning operation on new thread if remaining stack space becomes too small
- PR #2189³¹⁴⁰ - Adding callback taking index and future to when_each
- PR #2188³¹⁴¹ - Adding new example demonstrating receive_buffer
- PR #2187³¹⁴² - Mask 128-bit ints if CUDA is being used
- PR #2186³¹⁴³ - Make startup & shutdown functions unique_function
- PR #2185³¹⁴⁴ - Fixing logging output not to cause hang on shutdown
- PR #2184³¹⁴⁵ - Allowing component clients as action return types
- Issue #2183³¹⁴⁶ - Enabling logging output causes hang on shutdown
- Issue #2182³¹⁴⁷ - 1d_stencil seg fault
- Issue #2181³¹⁴⁸ - Setting small stack size does not change default
- PR #2180³¹⁴⁹ - Changing default bind mode to balanced
- PR #2179³¹⁵⁰ - adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2177³¹⁵¹ - Fixing 2176
- Issue #2176³¹⁵² - Launch process test fails on OSX
- PR #2175³¹⁵³ - Fix unbalanced config/warnings includes, add some new ones
- PR #2174³¹⁵⁴ - Fix test categorization : regression not unit
- Issue #2172³¹⁵⁵ - Different performance results
- Issue #2171³¹⁵⁶ - “negative entry in reference count table” running octotiger on 32 nodes on queenbee
- Issue #2170³¹⁵⁷ - Error while compiling on Mac + boost 1.60
- PR #2168³¹⁵⁸ - Fixing problems with is_bitwise_serializable
- Issue #2167³¹⁵⁹ - startup & shutdown function should accept unique_function
- Issue #2166³¹⁶⁰ - Simple receive_buffer example
- PR #2165³¹⁶¹ - Fix wait all

³¹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2190>

³¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2189>

³¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2188>

³¹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2187>

³¹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2186>

³¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2185>

³¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2184>

³¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2183>

³¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2182>

³¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2181>

³¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2180>

³¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2179>

³¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2177>

³¹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2176>

³¹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2175>

³¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2174>

³¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2172>

³¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2171>

³¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2170>

³¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2168>

³¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2167>

³¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2166>

³¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2165>

- PR #2164³¹⁶² - Fix wait all
- PR #2163³¹⁶³ - Fix some typos in config tests
- PR #2162³¹⁶⁴ - Improve #includes
- PR #2160³¹⁶⁵ - Add inspect check for missing #include <list>
- PR #2159³¹⁶⁶ - Add missing finalize call to stop test hanging
- PR #2158³¹⁶⁷ - Algo fixes
- PR #2157³¹⁶⁸ - Stack check
- Issue #2156³¹⁶⁹ - OSX reports stack space incorrectly (generic context coroutines)
- Issue #2155³¹⁷⁰ - Race condition suspected in runtime
- PR #2154³¹⁷¹ - Replace boost::detail::atomic_count with the new util::atomic_count
- PR #2153³¹⁷² - Fix stack overflow on OSX
- PR #2152³¹⁷³ - Define is_bitwise_serializable as is_trivially_copyable when available
- PR #2151³¹⁷⁴ - Adding missing <cstring> for std::mem* functions
- Issue #2150³¹⁷⁵ - Unable to use component clients as action return types
- PR #2149³¹⁷⁶ - std::memmove copies bytes, use bytes* sizeof(type) when copying larger types
- PR #2146³¹⁷⁷ - Adding customization point for parallel copy/move
- PR #2145³¹⁷⁸ - Applying changes to address warnings issued by latest version of PVS Studio
- Issue #2148³¹⁷⁹ - hpx::parallel::copy is broken after trivially copyable changes
- PR #2144³¹⁸⁰ - Some minor tweaks to compute prototype
- PR #2143³¹⁸¹ - Added Boost version support information over OSX platform
- PR #2142³¹⁸² - Fixing memory leak in example
- PR #2141³¹⁸³ - Add missing specializations in execution policies
- PR #2139³¹⁸⁴ - This PR fixes a few problems reported by Clang's Undefined Behavior sanitizer

³¹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2164>

³¹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2163>

³¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2162>

³¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2160>

³¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2159>

³¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2158>

³¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2157>

³¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2156>

³¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2155>

³¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2154>

³¹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2153>

³¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2152>

³¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2151>

³¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2150>

³¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2149>

³¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2146>

³¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2145>

³¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2148>

³¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2144>

³¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2143>

³¹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2142>

³¹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2141>

³¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2139>

- PR #2138³¹⁸⁵ - Revert “Adding fedora docs”
- PR #2136³¹⁸⁶ - Removed double semicolon
- PR #2135³¹⁸⁷ - Add deprecated #include check for hpx_fwd.hpp
- PR #2134³¹⁸⁸ - Resolved memory leak in stencil_8
- PR #2133³¹⁸⁹ - Replace uses of boost pointer containers
- PR #2132³¹⁹⁰ - Removing unused typedef
- PR #2131³¹⁹¹ - Add several include checks for std facilities
- PR #2130³¹⁹² - Fixing parcel compression, adding test
- PR #2129³¹⁹³ - Fix invalid attribute warnings
- Issue #2128³¹⁹⁴ - hpx::init seems to segfault
- PR #2127³¹⁹⁵ - Making executor_traits N-nary
- PR #2126³¹⁹⁶ - GCC 4.6 fails to deduce the correct type in lambda
- PR #2125³¹⁹⁷ - Making parcel coalescing test actually test something
- Issue #2124³¹⁹⁸ - Make a testcase for parcel compression
- Issue #2123³¹⁹⁹ - hpx/hpx/runtime/appplier_fwd.hpp - Multiple defined types
- Issue #2122³²⁰⁰ - Exception in primary_namespace::resolve_free_list
- Issue #2121³²⁰¹ - Possible memory leak in 1d_stencil_8
- PR #2120³²⁰² - Fixing 2119
- Issue #2119³²⁰³ - reduce_by_key compilation problems
- Issue #2118³²⁰⁴ - Premature unwrapping of boost::ref’ed arguments
- PR #2117³²⁰⁵ - Added missing initializer on last constructor for thread_description
- PR #2116³²⁰⁶ - Use a lightweight bind implementation when no placeholders are given
- PR #2115³²⁰⁷ - Replace boost::shared_ptr with std::shared_ptr

³¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2138>

³¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2136>

³¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2135>

³¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2134>

³¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2133>

³¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2132>

³¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2131>

³¹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2130>

³¹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2129>

³¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2128>

³¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2127>

³¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2126>

³¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2125>

³¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2124>

³¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2123>

³²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2122>

³²⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/2121>

³²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2120>

³²⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2119>

³²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2118>

³²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2117>

³²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2116>

³²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2115>

- PR #2114³²⁰⁸ - Adding hook functions for executor_parameter_traits supporting timers
- Issue #2113³²⁰⁹ - Compilation error with gcc version 4.9.3 (MacPorts gcc49 4.9.3_0)
- PR #2112³²¹⁰ - Replace uses of safe_bool with explicit operator bool
- Issue #2111³²¹¹ - Compilation error on QT example
- Issue #2110³²¹² - Compilation error when passing non-future argument to unwrapped continuation in dataflow
- Issue #2109³²¹³ - Warning while compiling hpx
- Issue #2109³²¹⁴ - Stack trace of last bug causing issues with octotiger
- Issue #2108³²¹⁵ - Stack trace of last bug causing issues with octotiger
- PR #2107³²¹⁶ - Making sure that a missing parcel_coalescing module does not cause startup exceptions
- PR #2106³²¹⁷ - Stop using hpx_fwd.hpp
- Issue #2105³²¹⁸ - coalescing plugin handler is not optional any more
- Issue #2104³²¹⁹ - Make executor_traits N-nary
- Issue #2103³²²⁰ - Build error with octotiger and hpx commit e657426d
- PR #2102³²²¹ - Combining thread data storage
- PR #2101³²²² - Added repartition version of 1d stencil that uses any performance counter
- PR #2100³²²³ - Drop obsolete TR1 result_of protocol
- PR #2099³²²⁴ - Replace uses of boost::bind with util::bind
- PR #2098³²²⁵ - Deprecated inspect checks
- PR #2097³²²⁶ - Reduce by key, extends #1141
- PR #2096³²²⁷ - Moving local cache from external to hpx/util
- PR #2095³²²⁸ - Bump minimum required Boost to 1.50.0
- PR #2094³²²⁹ - Add include checks for several Boost utilities
- Issue #2093³²³⁰ - /.../local_cache.hpp(89): error #303: explicit type is missing (“int” assumed)

³²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2114>

³²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2113>

³²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2112>

³²¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2111>

³²¹² <https://github.com/STELLAR-GROUP/hpx/issues/2110>

³²¹³ <https://github.com/STELLAR-GROUP/hpx/issues/2109>

³²¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2109>

³²¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2108>

³²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2107>

³²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2106>

³²¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2105>

³²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2104>

³²²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2103>

³²²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2102>

³²²² <https://github.com/STELLAR-GROUP/hpx/pull/2101>

³²²³ <https://github.com/STELLAR-GROUP/hpx/pull/2100>

³²²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2099>

³²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2098>

³²²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2097>

³²²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2096>

³²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2095>

³²²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2094>

³²³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2093>

- PR #2091³²³¹ - Fix for Raspberry pi build
- PR #2090³²³² - Fix storage size for util::function<>
- PR #2089³²³³ - Fix #2088
- Issue #2088³²³⁴ - More verbose output from cmake configuration
- PR #2087³²³⁵ - Making sure init_globally always executes hpx_main
- Issue #2086³²³⁶ - Race condition with recent HPX
- PR #2085³²³⁷ - Adding #include checker
- PR #2084³²³⁸ - Replace boost lock types with standard library ones
- PR #2083³²³⁹ - Simplify packaged task
- PR #2082³²⁴⁰ - Updating APEX version for testing
- PR #2081³²⁴¹ - Cleanup exception headers
- PR #2080³²⁴² - Make call_once variadic
- Issue #2079³²⁴³ - With GNU C++, line 85 of hpx/config/version.hpp causes link failure when linking application
- Issue #2078³²⁴⁴ - Simple test fails with _GLIBCXX_DEBUG defined
- PR #2077³²⁴⁵ - Instantiate board in nqueen client
- PR #2076³²⁴⁶ - Moving coalescing registration to TUs
- PR #2075³²⁴⁷ - Fixed some documentation typos
- PR #2074³²⁴⁸ - Adding flush-mode to message handler flush
- PR #2073³²⁴⁹ - Fixing performance regression introduced lately
- PR #2072³²⁵⁰ - Refactor local::condition_variable
- PR #2071³²⁵¹ - Timer based on boost::asio::deadline_timer
- PR #2070³²⁵² - Refactor tuple based functionality
- PR #2069³²⁵³ - Fixed typos

³²³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2091>

³²³² <https://github.com/STELLAR-GROUP/hpx/pull/2090>

³²³³ <https://github.com/STELLAR-GROUP/hpx/pull/2089>

³²³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2088>

³²³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2087>

³²³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2086>

³²³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2085>

³²³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2084>

³²³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2083>

³²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2082>

³²⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2081>

³²⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2080>

³²⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2079>

³²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2078>

³²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2077>

³²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2076>

³²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2075>

³²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2074>

³²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2073>

³²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2072>

³²⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2071>

³²⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2070>

³²⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2069>

- Issue #2068³²⁵⁴ - Seg fault with octotiger
- PR #2067³²⁵⁵ - Algorithm cleanup
- PR #2066³²⁵⁶ - Split credit fixes
- PR #2065³²⁵⁷ - Rename HPX_MOVABLE_BUT_NOT_COPYABLE to HPX_MOVABLE_ONLY
- PR #2064³²⁵⁸ - Fixed some typos in docs
- PR #2063³²⁵⁹ - Adding example demonstrating template components
- Issue #2062³²⁶⁰ - Support component templates
- PR #2061³²⁶¹ - Replace some uses of lexical_cast<string> with C++11 std::to_string
- PR #2060³²⁶² - Replace uses of boost::noncopyable with HPX_NON_COPYABLE
- PR #2059³²⁶³ - Adding missing for_loop algorithms
- PR #2058³²⁶⁴ - Move several definitions to more appropriate headers
- PR #2057³²⁶⁵ - Simplify assert_owns_lock and ignore_while_checking
- PR #2056³²⁶⁶ - Replacing std::result_of with util::result_of
- PR #2055³²⁶⁷ - Fix process launching/connecting back
- PR #2054³²⁶⁸ - Add a forwarding coroutine header
- PR #2053³²⁶⁹ - Replace uses of boost::unordered_map with std::unordered_map
- PR #2052³²⁷⁰ - Rewrite tuple unwrap
- PR #2050³²⁷¹ - Replace uses of BOOST_SCOPED_ENUM with C++11 scoped enums
- PR #2049³²⁷² - Attempt to narrow down split_credit problem
- PR #2048³²⁷³ - Fixing gcc startup hangs
- PR #2047³²⁷⁴ - Fixing when_xxx and wait_xxx for MSVC12
- PR #2046³²⁷⁵ - adding persistent_auto_chunk_size and related tests for for_each
- PR #2045³²⁷⁶ - Fixing HPX_HAVE_THREAD_BACKTRACE_DEPTH build time configuration

³²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2068>

³²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2067>

³²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2066>

³²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2065>

³²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2064>

³²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2063>

³²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2062>

³²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2061>

³²⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2060>

³²⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2059>

³²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2058>

³²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2057>

³²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2056>

³²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2055>

³²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2054>

³²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2053>

³²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2052>

³²⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2050>

³²⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2049>

³²⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2048>

³²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2047>

³²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2046>

³²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2045>

- PR #2044³²⁷⁷ - Adding missing service executor types
- PR #2043³²⁷⁸ - Removing ambiguous definitions for is_future_range and future_range_traits
- PR #2042³²⁷⁹ - Clarify that HPX builds can use (much) more than 2GB per process
- PR #2041³²⁸⁰ - Changing future_iterator_traits to support pointers
- Issue #2040³²⁸¹ - Improve documentation memory usage warning?
- PR #2039³²⁸² - Coroutine cleanup
- PR #2038³²⁸³ - Fix cmake policy CMP0042 warning MACOSX_RPATH
- PR #2037³²⁸⁴ - Avoid redundant specialization of [**unique**]function_nonser
- PR #2036³²⁸⁵ - nvcc dies with an internal error upon pushing/popping warnings inside templates
- Issue #2035³²⁸⁶ - Use a less restrictive iterator definition in hpx::lcos::detail::future_iterator_traits
- PR #2034³²⁸⁷ - Fixing compilation error with thread queue wait time performance counter
- Issue #2033³²⁸⁸ - Compilation error when compiling with thread queue waittime performance counter
- Issue #2032³²⁸⁹ - Ambiguous template instantiation for is_future_range and future_range_traits.
- PR #2031³²⁹⁰ - Don't restart timer on every incoming parcel
- PR #2030³²⁹¹ - Unify handling of execution policies in parallel algorithms
- PR #2029³²⁹² - Make pkg-config .pc files use .dylib on OSX
- PR #2028³²⁹³ - Adding process component
- PR #2027³²⁹⁴ - Making check for compiler compatibility independent on compiler path
- PR #2025³²⁹⁵ - Fixing inspect tool
- PR #2024³²⁹⁶ - Intel13 removal
- PR #2023³²⁹⁷ - Fix errors related to older boost versions and parameter pack expansions in lambdas
- Issue #2022³²⁹⁸ - gmake fail: “No rule to make target /usr/lib46/libboost_context-mt.so”
- PR #2021³²⁹⁹ - Added Sudoku example

³²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2044>

³²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2043>

³²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2042>

³²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2041>

³²⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/2040>

³²⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2039>

³²⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2038>

³²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2037>

³²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2036>

³²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2035>

³²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2034>

³²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2033>

³²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2032>

³²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2031>

³²⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2030>

³²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2029>

³²⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2028>

³²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2027>

³²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2025>

³²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2024>

³²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2023>

³²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2022>

³²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2021>

- Issue #2020³³⁰⁰ - Make errors related to init_globally.cpp example while building HPX out of the box
- PR #2019³³⁰¹ - Fixed some compilation and cmake errors encountered in nqueen example
- PR #2018³³⁰² - For loop algorithms
- PR #2017³³⁰³ - Non-recursive at_index implementation
- Issue #2016³³⁰⁴ - Add index-based for-loops
- Issue #2015³³⁰⁵ - Change default bind-mode to balanced
- PR #2014³³⁰⁶ - Fixed dataflow if invoked action returns a future
- PR #2013³³⁰⁷ - Fixing compilation issues with external example
- PR #2012³³⁰⁸ - Added Sierpinski Triangle example
- Issue #2011³³⁰⁹ - Compilation error while running sample hello_world_component code
- PR #2010³³¹⁰ - Segmented move implemented for hpx::vector
- Issue #2009³³¹¹ - pkg-config order incorrect on 14.04 / GCC 4.8
- Issue #2008³³¹² - Compilation error in dataflow of action returning a future
- PR #2007³³¹³ - Adding new performance counter exposing overall scheduler time
- PR #2006³³¹⁴ - Function includes
- PR #2005³³¹⁵ - Adding an example demonstrating how to initialize HPX from a global object
- PR #2004³³¹⁶ - Fixing 2000
- PR #2003³³¹⁷ - Adding generation parameter to gather to enable using it more than once
- PR #2002³³¹⁸ - Turn on position independent code to solve link problem with hpx_init
- Issue #2001³³¹⁹ - Gathering more than once segfaults
- Issue #2000³³²⁰ - Undefined reference to hpx::assertion_failed
- Issue #1999³³²¹ - Seg fault in hpx::lcos::base_lco_with_value<*>::set_value_nonvirt() when running octo-tiger
- PR #1998³³²² - Detect unknown command line options

³³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2020>

³³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2019>

³³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2018>

³³⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2017>

³³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2016>

³³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2015>

³³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2014>

³³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2013>

³³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2012>

³³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2011>

³³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2010>

³³¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2009>

³³¹² <https://github.com/STELLAR-GROUP/hpx/issues/2008>

³³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2007>

³³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2006>

³³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2005>

³³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2004>

³³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2003>

³³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2002>

³³¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2001>

³³²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2000>

³³²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1999>

³³²² <https://github.com/STELLAR-GROUP/hpx/pull/1998>

- PR #1997³³²³ - Extending thread description
- PR #1996³³²⁴ - Adding natvis files to solution (MSVC only)
- Issue #1995³³²⁵ - Command line handling does not produce error
- PR #1994³³²⁶ - Possible missing include in test_utils.hpp
- PR #1993³³²⁷ - Add missing LANGUAGES tag to a hpx_add_compile_flag_if_available() call in CMake-Lists.txt
- PR #1992³³²⁸ - Fixing shared_executor_test
- PR #1991³³²⁹ - Making sure the winsock library is properly initialized
- PR #1990³³³⁰ - Fixing bind_test placeholder ambiguity coming from boost-1.60
- PR #1989³³³¹ - Performance tuning
- PR #1987³³³² - Make configurable size of internal storage in util::function
- PR #1986³³³³ - AGAS Refactoring+1753 Cache mods
- PR #1985³³³⁴ - Adding missing task_block::run() overload taking an executor
- PR #1984³³³⁵ - Adding an optimized LRU Cache implementation (for AGAS)
- PR #1983³³³⁶ - Avoid invoking migration table look up for all objects
- PR #1981³³³⁷ - Replacing uintptr_t (which is not defined everywhere) with std::size_t
- PR #1980³³³⁸ - Optimizing LCO continuations
- PR #1979³³³⁹ - Fixing Cori
- PR #1978³³⁴⁰ - Fix test check that got broken in hasty fix to memory overflow
- PR #1977³³⁴¹ - Refactor action traits
- PR #1976³³⁴² - Fixes typo in README.rst
- PR #1975³³⁴³ - Reduce size of benchmark timing arrays to fix test failures
- PR #1974³³⁴⁴ - Add action to update data owned by the partitioned_vector component
- PR #1972³³⁴⁵ - Adding partitioned_vector SPMD example

³³²³ <https://github.com/STELLAR-GROUP/hpx/pull/1997>

³³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1996>

³³²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1995>

³³²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1994>

³³²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1993>

³³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1992>

³³²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1991>

³³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1990>

³³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1989>

³³³² <https://github.com/STELLAR-GROUP/hpx/pull/1987>

³³³³ <https://github.com/STELLAR-GROUP/hpx/pull/1986>

³³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1985>

³³³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1984>

³³³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1983>

³³³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1981>

³³³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1980>

³³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1979>

³³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1978>

³³⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1977>

³³⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1976>

³³⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1975>

³³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1974>

³³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1972>

- PR #1971³³⁴⁶ - Fixing 1965
- PR #1970³³⁴⁷ - Papi fixes
- PR #1969³³⁴⁸ - Fixing continuation recursions to not depend on fixed amount of recursions
- PR #1968³³⁴⁹ - More segmented algorithms
- Issue #1967³³⁵⁰ - Simplify component implementations
- PR #1966³³⁵¹ - Migrate components
- Issue #1964³³⁵² - fatal error: ‘boost/lockfree/detail/branch_hints.hpp’ file not found
- Issue #1962³³⁵³ - parallel:copy_if has race condition when used on in place arrays
- PR #1963³³⁵⁴ - Fixing Static Parcelport initialization
- PR #1961³³⁵⁵ - Fix function target
- Issue #1960³³⁵⁶ - Papi counters don’t reset
- PR #1959³³⁵⁷ - Fixing 1958
- Issue #1958³³⁵⁸ - inclusive_scan gives incorrect results with non-commutative operator
- PR #1957³³⁵⁹ - Fixing #1950
- PR #1956³³⁶⁰ - Sort by key example
- PR #1955³³⁶¹ - Adding regression test for #1946: Hang in wait_all() in distributed run
- Issue #1954³³⁶² - HPX releases should not use -Werror
- PR #1953³³⁶³ - Adding performance analysis for AGAS cache
- PR #1952³³⁶⁴ - Adapting test for explicit variadics to fail for gcc 4.6
- PR #1951³³⁶⁵ - Fixing memory leak
- Issue #1950³³⁶⁶ - Simplify external builds
- PR #1949³³⁶⁷ - Fixing yet another lock that is being held during suspension
- PR #1948³³⁶⁸ - Fixed container algorithms for Intel

³³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1971>

³³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1970>

³³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1969>

³³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1968>

³³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1967>

³³⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1966>

³³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1964>

³³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1962>

³³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1963>

³³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1961>

³³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1960>

³³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1959>

³³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1958>

³³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1957>

³³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1956>

³³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1955>

³³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1954>

³³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1953>

³³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1952>

³³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1951>

³³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1950>

³³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1949>

³³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1948>

- PR #1947³³⁶⁹ - Adding workaround for tagged_tuple
- Issue #1946³³⁷⁰ - Hang in wait_all() in distributed run
- PR #1945³³⁷¹ - Fixed container algorithm tests
- Issue #1944³³⁷² - assertion ‘p.destination_locality() == hpx::get_locality()’ failed
- PR #1943³³⁷³ - Fix a couple of compile errors with clang
- PR #1942³³⁷⁴ - Making parcel coalescing functional
- Issue #1941³³⁷⁵ - Re-enable parcel coalescing
- PR #1940³³⁷⁶ - Touching up make_future
- PR #1939³³⁷⁷ - Fixing problems in over-subscription management in the resource manager
- PR #1938³³⁷⁸ - Removing use of unified Boost.Thread header
- PR #1937³³⁷⁹ - Cleaning up the use of Boost.Accumulator headers
- PR #1936³³⁸⁰ - Making sure interval timer is started for aggregating performance counters
- PR #1935³³⁸¹ - Tagged results
- PR #1934³³⁸² - Fix remote async with deferred launch policy
- Issue #1933³³⁸³ - Floating point exception in statistics_counter<boost::accumulators::tag::mean>::get_c...
- PR #1932³³⁸⁴ - Removing superfluous includes of boost/lockfree/detail/branch_hints.hpp
- PR #1931³³⁸⁵ - fix compilation with clang 3.8.0
- Issue #1930³³⁸⁶ - Missing online documentation for HPX 0.9.11
- PR #1929³³⁸⁷ - LWG2485: get() should be overloaded for const tuple&&
- PR #1928³³⁸⁸ - Revert “Using ninja for circle-ci builds”
- PR #1927³³⁸⁹ - Using ninja for circle-ci builds
- PR #1926³³⁹⁰ - Fixing serialization of std::array
- Issue #1925³³⁹¹ - Issues with static HPX libraries

³³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1947>

³³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1946>

³³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1945>

³³⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1944>

³³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1943>

³³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1942>

³³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1941>

³³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1940>

³³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1939>

³³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1938>

³³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1937>

³³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1936>

³³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1935>

³³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1934>

³³⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1933>

³³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1932>

³³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1931>

³³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1930>

³³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1929>

³³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1928>

³³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1927>

³³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1926>

³³⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1925>

- Issue #1924³³⁹² - Performance degrading over time
- Issue #1923³³⁹³ - serialization of std::array appears broken in latest commit
- PR #1922³³⁹⁴ - Container algorithms
- PR #1921³³⁹⁵ - Tons of smaller quality improvements
- Issue #1920³³⁹⁶ - Seg fault in hpx::serialization::output_archive::add_gid when running octotiger
- Issue #1919³³⁹⁷ - Intel 15 compiler bug preventing HPX build
- PR #1918³³⁹⁸ - Address sanitizer fixes
- PR #1917³³⁹⁹ - Fixing compilation problems of parallel::sort with Intel compilers
- PR #1916³⁴⁰⁰ - Making sure code compiles if HPX_WITH_HWLOC=Off
- Issue #1915³⁴⁰¹ - max_cores undefined if HPX_WITH_HWLOC=Off
- PR #1913³⁴⁰² - Add utility member functions for partitioned_vector
- PR #1912³⁴⁰³ - Adding support for invoking actions to dataflow
- PR #1911³⁴⁰⁴ - Adding first batch of container algorithms
- PR #1910³⁴⁰⁵ - Keep cmake_module_path
- PR #1909³⁴⁰⁶ - Fix mpirun with pbs
- PR #1908³⁴⁰⁷ - Changing parallel::sort to return the last iterator as proposed by N4560
- PR #1907³⁴⁰⁸ - Adding a minimum version for Open MPI
- PR #1906³⁴⁰⁹ - Updates to the Release Procedure
- PR #1905³⁴¹⁰ - Fixing #1903
- PR #1904³⁴¹¹ - Making sure std containers are cleared before serialization loads data
- Issue #1903³⁴¹² - When running octotiger, I get: assertion '(*new_gids_)[gid].size() == 1' failed: HPX(assertion_failure)
- Issue #1902³⁴¹³ - Immediate crash when running hpx/octotiger with _GLIBCXX_DEBUG defined.
- PR #1901³⁴¹⁴ - Making non-serializable classes non-serializable

³³⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1924>

³³⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1923>

³³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1922>

³³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1921>

³³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1920>

³³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1919>

³³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1918>

³³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1917>

³⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1916>

³⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1915>

³⁴⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1913>

³⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1912>

³⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1911>

³⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1910>

³⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1909>

³⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1908>

³⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1907>

³⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1906>

³⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1905>

³⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1904>

³⁴¹² <https://github.com/STELLAR-GROUP/hpx/issues/1903>

³⁴¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1902>

³⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1901>

- Issue #1900³⁴¹⁵ - Two possible issues with std::list serialization
- PR #1899³⁴¹⁶ - Fixing a problem with credit splitting as revealed by #1898
- Issue #1898³⁴¹⁷ - Accessing component from locality where it was not created segfaults
- PR #1897³⁴¹⁸ - Changing parallel::sort to return the last iterator as proposed by N4560
- Issue #1896³⁴¹⁹ - version 1.0?
- Issue #1895³⁴²⁰ - Warning comment on numa_allocator is not very clear
- PR #1894³⁴²¹ - Add support for compilers that have thread_local
- PR #1893³⁴²² - Fixing 1890
- PR #1892³⁴²³ - Adds typed future_type for executor_traits
- PR #1891³⁴²⁴ - Fix wording in certain parallel algorithm docs
- Issue #1890³⁴²⁵ - Invoking papi counters give segfault
- PR #1889³⁴²⁶ - Fixing problems as reported by clang-check
- PR #1888³⁴²⁷ - WIP parallel is_heap
- PR #1887³⁴²⁸ - Fixed resetting performance counters related to idle-rate, etc
- Issue #1886³⁴²⁹ - Run hpx with qsub does not work
- PR #1885³⁴³⁰ - Warning cleaning pass
- PR #1884³⁴³¹ - Add missing parallel algorithm header
- PR #1883³⁴³² - Add feature test for thread_local on Clang for TLS
- PR #1882³⁴³³ - Fix some redundant qualifiers
- Issue #1881³⁴³⁴ - Unable to compile Octotiger using HPX and Intel MPI on SuperMIC
- Issue #1880³⁴³⁵ - clang with libc++ on Linux needs TLS case
- PR #1879³⁴³⁶ - Doc fixes for #1868
- PR #1878³⁴³⁷ - Simplify functions

³⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1900>

³⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1899>

³⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1898>

³⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1897>

³⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1896>

³⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1895>

³⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1894>

³⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/1893>

³⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/1892>

³⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1891>

³⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1890>

³⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1889>

³⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1888>

³⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1887>

³⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1886>

³⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1885>

³⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1884>

³⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/1883>

³⁴³³ <https://github.com/STELLAR-GROUP/hpx/pull/1882>

³⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1881>

³⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1880>

³⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1879>

³⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1878>

- PR #1877³⁴³⁸ - Removing most usage of Boost.Config
- PR #1876³⁴³⁹ - Add missing parallel algorithms to algorithm.hpp
- PR #1875³⁴⁴⁰ - Simplify callables
- PR #1874³⁴⁴¹ - Address long standing FIXME on using std::unique_ptr with incomplete types
- PR #1873³⁴⁴² - Fixing 1871
- PR #1872³⁴⁴³ - Making sure PBS environment uses specified node list even if no PBS_NODEFILE env is available
- Issue #1871³⁴⁴⁴ - Fortran checks should be optional
- PR #1870³⁴⁴⁵ - Touch local::mutex
- PR #1869³⁴⁴⁶ - Documentation refactoring based off #1868
- PR #1867³⁴⁴⁷ - Embrace static_assert
- PR #1866³⁴⁴⁸ - Fix #1803 with documentation refactoring
- PR #1865³⁴⁴⁹ - Setting OUTPUT_NAME as target properties
- PR #1863³⁴⁵⁰ - Use SYSTEM for boost includes
- PR #1862³⁴⁵¹ - Minor cleanups
- PR #1861³⁴⁵² - Minor Corrections for Release
- PR #1860³⁴⁵³ - Fixing hpx gdb script
- Issue #1859³⁴⁵⁴ - reset_active_counters resets times and thread counts before some of the counters are evaluated
- PR #1858³⁴⁵⁵ - Release V0.9.11
- PR #1857³⁴⁵⁶ - removing diskperf example from 9.11 release
- PR #1856³⁴⁵⁷ - fix return in packaged_task_base::reset()
- Issue #1842³⁴⁵⁸ - Install error: file INSTALL cannot find libhpx_parcel_coalescing.so.0.9.11
- PR #1839³⁴⁵⁹ - Adding fedora docs
- PR #1824³⁴⁶⁰ - Changing version on master to V0.9.12

³⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1877>

³⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1876>

³⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1875>

³⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1874>

³⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1873>

³⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1872>

³⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1871>

³⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1870>

³⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1869>

³⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1867>

³⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1866>

³⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1865>

³⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1863>

³⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1862>

³⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1861>

³⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1860>

³⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1859>

³⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1858>

³⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1857>

³⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1856>

³⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1842>

³⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1839>

³⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1824>

- PR #1818³⁴⁶¹ - Fixing #1748
- Issue #1815³⁴⁶² - seg fault in AGAS
- Issue #1803³⁴⁶³ - wait_all documentation
- Issue #1796³⁴⁶⁴ - Outdated documentation to be revised
- Issue #1759³⁴⁶⁵ - glibc munmap_chunk or free(): invalid pointer on SuperMIC
- Issue #1753³⁴⁶⁶ - HPX performance degrades with time since execution begins
- Issue #1748³⁴⁶⁷ - All public HPX headers need to be self contained
- PR #1719³⁴⁶⁸ - How to build HPX with Visual Studio
- Issue #1684³⁴⁶⁹ - Race condition when using --hpx:connect?
- PR #1658³⁴⁷⁰ - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1641³⁴⁷¹ - Generic client
- Issue #1632³⁴⁷² - heartbeat example fails on separate nodes
- PR #1603³⁴⁷³ - Adds preferred namespace check to inspect tool
- Issue #1559³⁴⁷⁴ - Extend inspect tool
- Issue #1523³⁴⁷⁵ - Remote async with deferred launch policy never executes
- Issue #1472³⁴⁷⁶ - Serialization issues
- Issue #1457³⁴⁷⁷ - Implement N4392: C++ Latches and Barriers
- PR #1444³⁴⁷⁸ - Enabling usage of moveonly types for component construction
- Issue #1407³⁴⁷⁹ - The Intel 13 compiler has failing unit tests
- Issue #1405³⁴⁸⁰ - Allow component constructors to take movable only types
- Issue #1265³⁴⁸¹ - Enable dataflow() to be usable with actions
- Issue #1236³⁴⁸² - NUMA aware allocators
- Issue #802³⁴⁸³ - Fix Broken Examples

³⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1818>

³⁴⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1815>

³⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1803>

³⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1796>

³⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1759>

³⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1753>

³⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1748>

³⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1719>

³⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1684>

³⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1658>

³⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1641>

³⁴⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1632>

³⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1603>

³⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1559>

³⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1523>

³⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1472>

³⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1457>

³⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1444>

³⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1407>

³⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1405>

³⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1265>

³⁴⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1236>

³⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/802>

- Issue #559³⁴⁸⁴ - Add hpx::migrate facility
- Issue #449³⁴⁸⁵ - Make actions with template arguments usable and add documentation
- Issue #279³⁴⁸⁶ - Refactor addressing_service into a base class and two derived classes
- Issue #224³⁴⁸⁷ - Changing thread state metadata is not thread safe
- Issue #55³⁴⁸⁸ - Uniform syntax for enums should be implemented

2.10.14 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](#)³⁴⁸⁹ 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_`. It allows one to instantiate both, single component instances and multiple component instances. The placement of the created components can be controlled by special distribution policies. Please see the corresponding documentation outlining the use of `hpx::new_`.
- Introduced four new distribution policies which can be used with many API functions which traditionally expected to be used with a locality id. The new distribution policies are:
 - `hpx::components::default_distribution_policy` which tries to place multiple component instances as evenly as possible.

³⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/559>

³⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/449>

³⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/279>

³⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/224>

³⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/55>

³⁴⁸⁹ <https://circleci.com/gh/STELLAR-GROUP/hpx>

- `hpx::components::colocating_distribution_policy` which will refer to the locality where a given component instance is currently placed.
- `hpx::components::binpacking_distribution_policy` which will place multiple component instances as evenly as possible based on any performance counter.
- `hpx::components::target_distribution_policy` which allows one to represent a given locality in the context of a distribution policy.
- The new distribution policies can now be also used with `hpx::async`. This change also deprecates `hpx::async_colocated(id, ...)` which now is replaced by a distribution policy: `hpx::async(hpx::colocated(id), ...)`.
- The `hpx::vector` and `hpx::unordered_map` data structures can now be used with the new distribution policies as well.
- The parallel facility `hpx::parallel::task_region` has been renamed to `hpx::parallel::task_block` based on the changes in the corresponding standardization proposal [N4411³⁴⁹⁰](#).
- 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 [N4399³⁴⁹¹](#)).
- Added performance counters exposing data related to data transferred by input/output (filesystem) operations (thanks to Maciej Brodowicz).
- Added performance counters allowing to track the number of action invocations (local and remote invocations).
- Added new command line options `-hpx:print-counter-at` and `-hpx:reset-counters`.
- The `hpx::vector` component has been renamed to `hpx::partitioned_vector` to make it explicit that the underlying memory is not contiguous.
- Introduced a completely new and uniform higher-level parallelism API which is based on executors. All existing parallelism APIs have been adapted to this. We have added a large number of different executor types, such as a numa-aware executor, a this-thread executor, etc.
- Added support for the MingW toolchain on Windows (thanks to Eric Lemanissier).
- HPX now includes support for APEX, (Autonomic Performance Environment for eXascale). APEX is an instrumentation and software adaptation library that provides an interface to TAU profiling / tracing as well as runtime adaptation of HPX applications through policy definitions. For more information and documentation, please see <https://github.com/UO-OACISS/xpress-apex>. To enable APEX at configuration time, specify `-DHPX_WITH_APEX=On`. To also include support for TAU profiling, specify `-DHPX_WITH_TAU=On` and specify the `-DTAU_ROOT`, `-DTAU_ARCH` and `-DTAU_OPTIONS` cmake parameters.
- We have implemented many more of the *Using parallel algorithms*. Please see [Issue #1141³⁴⁹²](#) for the list of all available parallel algorithms (thanks to Daniel Bourgeois and John Biddiscombe for contributing their work).

³⁴⁹⁰ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

³⁴⁹¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html>

³⁴⁹² <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 announcement³⁴⁹³ on the hpx-users@stellar.cct.lsu.edu mailing list.
- The names used by cmake to influence various configuration options have been unified. The new naming scheme relies on all configuration constants to start with `HPX_WITH_...`, while the preprocessor constant which is used at build time starts with `HPX_HAVE_...`. For instance, the former cmake command line `-DHPX_MALLOC=...` now has to be specified a `-DHPX_WITH_MALLOC=...` and will cause the preprocessor constant `HPX_HAVE_MALLOC` to be defined. The actual name of the constant (i.e. `MALLOC`) has not changed. Please see the corresponding documentation for more details (*CMake variables used to configure HPX*).
- The `get_gid()` functions exposed by the component base classes `hpx::components::server::simple_component_base`, `hpx::components::server::managed_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`.

³⁴⁹³ <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 #1855³⁴⁹⁴ - Completely removing external/endian
- PR #1854³⁴⁹⁵ - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1853³⁴⁹⁶ - Updating CMake configuration to get correct version of TAU library
- PR #1852³⁴⁹⁷ - Fixing Performance Problems with MPI Parcelport
- PR #1851³⁴⁹⁸ - Fixing hpx_add_link_flag() and hpx_remove_link_flag()
- PR #1850³⁴⁹⁹ - Fixing 1836, adding parallel::sort
- PR #1849³⁵⁰⁰ - Fixing configuration for use of more than 64 cores
- PR #1848³⁵⁰¹ - Change default APEX version for release
- PR #1847³⁵⁰² - Fix client_base::then on release
- PR #1846³⁵⁰³ - Removing broken lcos::local::channel from release
- PR #1845³⁵⁰⁴ - Adding example demonstrating a possible safe-object implementation to release
- PR #1844³⁵⁰⁵ - Removing stubs from accumulator examples
- PR #1843³⁵⁰⁶ - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1841³⁵⁰⁷ - Fixing client_base<>::then
- PR #1840³⁵⁰⁸ - Adding example demonstrating a possible safe-object implementation
- PR #1838³⁵⁰⁹ - Update version rc1
- PR #1837³⁵¹⁰ - Removing broken lcos::local::channel
- PR #1835³⁵¹¹ - Adding explicit move constructor and assignment operator to hpx::lcos::promise
- PR #1834³⁵¹² - Making hpx::lcos::promise move-only
- PR #1833³⁵¹³ - Adding fedora docs
- Issue #1832³⁵¹⁴ - hpx::lcos::promise<> must be move-only

³⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1855>

³⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1854>

³⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1853>

³⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1852>

³⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1851>

³⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1850>

³⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1849>

³⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1848>

³⁵⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1847>

³⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1846>

³⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1845>

³⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1844>

³⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1843>

³⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1841>

³⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1840>

³⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1838>

³⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1837>

³⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1835>

³⁵¹² <https://github.com/STELLAR-GROUP/hpx/pull/1834>

³⁵¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1833>

³⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1832>

- PR #1831³⁵¹⁵ - Fixing resource manager gcc5.2
- PR #1830³⁵¹⁶ - Fix intel13
- PR #1829³⁵¹⁷ - Unbreaking thread test
- PR #1828³⁵¹⁸ - Fixing #1620
- PR #1827³⁵¹⁹ - Fixing a memory management issue for the Parquet application
- Issue #1826³⁵²⁰ - Memory management issue in hpx::lcos::promise
- PR #1825³⁵²¹ - Adding hpx::components::component and hpx::components::component_base
- PR #1823³⁵²² - Adding git commit id to circleci build
- PR #1822³⁵²³ - applying fixes suggested by clang 3.7
- PR #1821³⁵²⁴ - Hyperlink fixes
- PR #1820³⁵²⁵ - added parallel multi-locality sanity test
- PR #1819³⁵²⁶ - Fixing #1667
- Issue #1817³⁵²⁷ - Hyperlinks generated by inspect tool are wrong
- PR #1816³⁵²⁸ - Support hpxrx
- PR #1814³⁵²⁹ - Fix async to dispatch to the correct locality in all cases
- Issue #1813³⁵³⁰ - async(launch:..., action(), ...) always invokes locally
- PR #1812³⁵³¹ - fixed syntax error in CMakeLists.txt
- PR #1811³⁵³² - Agas optimizations
- PR #1810³⁵³³ - drop superfluous typedefs
- PR #1809³⁵³⁴ - Allow HPX to be used as an optional package in 3rd party code
- PR #1808³⁵³⁵ - Fixing #1723
- PR #1807³⁵³⁶ - Making sure resolve_localities does not hang during normal operation
- Issue #1806³⁵³⁷ - Spinlock no longer movable and deletes operator ‘=’, breaks MiniGhost

³⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1831>

³⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1830>

³⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1829>

³⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1828>

³⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1827>

³⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1826>

³⁵²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1825>

³⁵²² <https://github.com/STELLAR-GROUP/hpx/pull/1823>

³⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/1822>

³⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1821>

³⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1820>

³⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1819>

³⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1817>

³⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1816>

³⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1814>

³⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1813>

³⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1812>

³⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/1811>

³⁵³³ <https://github.com/STELLAR-GROUP/hpx/pull/1810>

³⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1809>

³⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1808>

³⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1807>

³⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1806>

- Issue #1804³⁵³⁸ - register_with_basename causes hangs
- PR #1801³⁵³⁹ - Enhanced the inspect tool to take user directly to the problem with hyperlinks
- Issue #1800³⁵⁴⁰ - Problems compiling application on smic
- PR #1799³⁵⁴¹ - Fixing cv exceptions
- PR #1798³⁵⁴² - Documentation refactoring & updating
- PR #1797³⁵⁴³ - Updating the activeharmony CMake module
- PR #1795³⁵⁴⁴ - Fixing cv
- PR #1794³⁵⁴⁵ - Fix connect with hpx::runtime_mode_connect
- PR #1793³⁵⁴⁶ - fix a wrong use of HPX_MAX_CPU_COUNT instead of HPX_HAVE_MAX_CPU_COUNT
- PR #1792³⁵⁴⁷ - Allow for default constructed parcel instances to be moved
- PR #1791³⁵⁴⁸ - Fix connect with hpx::runtime_mode_connect
- Issue #1790³⁵⁴⁹ - assertion action_.get () failed: HPX(assertion_failure) when running Octotiger with pull request 1786
- PR #1789³⁵⁵⁰ - Fixing discover_counter_types API function
- Issue #1788³⁵⁵¹ - connect with hpx::runtime_mode_connect
- Issue #1787³⁵⁵² - discover_counter_types not working
- PR #1786³⁵⁵³ - Changing addressing_service to use std::unordered_map instead of std::map
- PR #1785³⁵⁵⁴ - Fix is_iterator for container algorithms
- PR #1784³⁵⁵⁵ - Adding new command line options:
- PR #1783³⁵⁵⁶ - Minor changes for APEX support
- PR #1782³⁵⁵⁷ - Drop legacy forwarding action traits
- PR #1781³⁵⁵⁸ - Attempt to resolve the race between cv::wait_xxx and cv::notify_all
- PR #1780³⁵⁵⁹ - Removing serialize_sequence
- PR #1779³⁵⁶⁰ - Fixed #1501: hwloc configuration options are wrong for MIC

³⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1804>

³⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1801>

³⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1800>

³⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1799>

³⁵⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1798>

³⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1797>

³⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1795>

³⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1794>

³⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1793>

³⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1792>

³⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1791>

³⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1790>

³⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1789>

³⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1788>

³⁵⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1787>

³⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1786>

³⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1785>

³⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1784>

³⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1783>

³⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1782>

³⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1781>

³⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1780>

³⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1779>

- PR #1778³⁵⁶¹ - Removing ability to enable/disable parcel handling
- PR #1777³⁵⁶² - Completely removing stackless threads
- PR #1776³⁵⁶³ - Cleaning up util/plugin
- PR #1775³⁵⁶⁴ - Agas fixes
- PR #1774³⁵⁶⁵ - Action invocation count
- PR #1773³⁵⁶⁶ - replaced MSVC variable with WIN32
- PR #1772³⁵⁶⁷ - Fixing Problems in MPI parcelport and future serialization.
- PR #1771³⁵⁶⁸ - Fixing intel 13 compiler errors related to variadic template template parameters for `lcos::when_` tests
- PR #1770³⁵⁶⁹ - Forwarding decay to `std::`
- PR #1769³⁵⁷⁰ - Add more characters with special regex meaning to the existing patch
- PR #1768³⁵⁷¹ - Adding test for `receive_buffer`
- PR #1767³⁵⁷² - Making sure that uptime counter throws exception on any attempt to be reset
- PR #1766³⁵⁷³ - Cleaning up code related to throttling scheduler
- PR #1765³⁵⁷⁴ - Restricting `thread_data` to creating only with `intrusive_pointers`
- PR #1764³⁵⁷⁵ - Fixing 1763
- Issue #1763³⁵⁷⁶ - UB in `thread_data::operator delete`
- PR #1762³⁵⁷⁷ - Making sure all serialization registries/factories are unique
- PR #1761³⁵⁷⁸ - Fixed #1751: `hpx::future::wait_for` fails a simple test
- PR #1758³⁵⁷⁹ - Fixing #1757
- Issue #1757³⁵⁸⁰ - pinning not correct using `-hpx:bind`
- Issue #1756³⁵⁸¹ - compilation error with MinGW
- PR #1755³⁵⁸² - Making output serialization const-correct
- Issue #1753³⁵⁸³ - HPX performance degrades with time since execution begins

³⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1778>

³⁵⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1777>

³⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1776>

³⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1775>

³⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1774>

³⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1773>

³⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1772>

³⁵⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1771>

³⁵⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1770>

³⁵⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1769>

³⁵⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1768>

³⁵⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1767>

³⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1766>

³⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1765>

³⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1764>

³⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1763>

³⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1762>

³⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1761>

³⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1758>

³⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1757>

³⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1756>

³⁵⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1755>

³⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1753>

- Issue #1752³⁵⁸⁴ - Error in AGAS
- Issue #1751³⁵⁸⁵ - hpx::future::wait_for fails a simple test
- PR #1750³⁵⁸⁶ - Removing hpx_fwd.hpp includes
- PR #1749³⁵⁸⁷ - Simplify result_of and friends
- PR #1747³⁵⁸⁸ - Removed superfluous code from message_buffer.hpp
- PR #1746³⁵⁸⁹ - Tuple dependencies
- Issue #1745³⁵⁹⁰ - Broken when_some which takes iterators
- PR #1744³⁵⁹¹ - Refining archive interface
- PR #1743³⁵⁹² - Fixing when_all when only a single future is passed
- PR #1742³⁵⁹³ - Config includes
- PR #1741³⁵⁹⁴ - Os executors
- Issue #1740³⁵⁹⁵ - hpx::promise has some problems
- PR #1739³⁵⁹⁶ - Parallel composition with generic containers
- Issue #1738³⁵⁹⁷ - After building program and successfully linking to a version of hpx DHPX_DIR seems to be ignored
- Issue #1737³⁵⁹⁸ - Uptime problems
- PR #1736³⁵⁹⁹ - added convenience c-tor and begin()/end() to serialize_buffer
- PR #1735³⁶⁰⁰ - Config includes
- PR #1734³⁶⁰¹ - Fixed #1688: Add timer counters for tfunc_total and exec_total
- Issue #1733³⁶⁰² - Add unit test for hpx/lcos/local/receive_buffer.hpp
- PR #1732³⁶⁰³ - Renaming get_os_thread_count
- PR #1731³⁶⁰⁴ - Basename registration
- Issue #1730³⁶⁰⁵ - Use after move of thread_init_data
- PR #1729³⁶⁰⁶ - Rewriting channel based on new gate component

³⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1752>

³⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1751>

³⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1750>

³⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1749>

³⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1747>

³⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1746>

³⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1745>

³⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1744>

³⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1743>

³⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1742>

³⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1741>

³⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1740>

³⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1739>

³⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1738>

³⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1737>

³⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1736>

³⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1735>

³⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1734>

³⁶⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1733>

³⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1732>

³⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1731>

³⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1730>

³⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1729>

- PR #1728³⁶⁰⁷ - Fixing #1722
- PR #1727³⁶⁰⁸ - Fixing compile problems with apply_colocated
- PR #1726³⁶⁰⁹ - Apex integration
- PR #1725³⁶¹⁰ - fixed test timeouts
- PR #1724³⁶¹¹ - Renaming vector
- Issue #1723³⁶¹² - Drop support for intel compilers and gcc 4.4. based standard libs
- Issue #1722³⁶¹³ - Add support for detecting non-ready futures before serialization
- PR #1721³⁶¹⁴ - Unifying parallel executors, initializing from launch policy
- PR #1720³⁶¹⁵ - dropped superfluous typedef
- Issue #1718³⁶¹⁶ - Windows 10 x64, VS 2015 - Unknown CMake command “add_hpx_pseudo_target”.
- PR #1717³⁶¹⁷ - Timed executor traits for thread-executors
- PR #1716³⁶¹⁸ - serialization of arrays didn’t work with non-pod types. fixed
- PR #1715³⁶¹⁹ - List serialization
- PR #1714³⁶²⁰ - changing misspellings
- PR #1713³⁶²¹ - Fixed distribution policy executors
- PR #1712³⁶²² - Moving library detection to be executed after feature tests
- PR #1711³⁶²³ - Simplify parcel
- PR #1710³⁶²⁴ - Compile only tests
- PR #1709³⁶²⁵ - Implemented timed executors
- PR #1708³⁶²⁶ - Implement parallel::executor_traits for thread-executors
- PR #1707³⁶²⁷ - Various fixes to threads::executors to make custom schedulers work
- PR #1706³⁶²⁸ - Command line option –hpx:cores does not work as expected
- Issue #1705³⁶²⁹ - command line option –hpx:cores does not work as expected

³⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1728>

³⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1727>

³⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1726>

³⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1725>

³⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1724>

³⁶¹² <https://github.com/STELLAR-GROUP/hpx/issues/1723>

³⁶¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1722>

³⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1721>

³⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1720>

³⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1718>

³⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1717>

³⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1716>

³⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1715>

³⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1714>

³⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1713>

³⁶²² <https://github.com/STELLAR-GROUP/hpx/pull/1712>

³⁶²³ <https://github.com/STELLAR-GROUP/hpx/pull/1711>

³⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1710>

³⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1709>

³⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1708>

³⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1707>

³⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1706>

³⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1705>

- PR #1704³⁶³⁰ - vector deserialization is speeded up a little
- PR #1703³⁶³¹ - Fixing shared_mutes
- Issue #1702³⁶³² - Shared_mutex does not compile with no_mutex cond_var
- PR #1701³⁶³³ - Add distribution_policy_executor
- PR #1700³⁶³⁴ - Executor parameters
- PR #1699³⁶³⁵ - Readers writer lock
- PR #1698³⁶³⁶ - Remove leftovers
- PR #1697³⁶³⁷ - Fixing held locks
- PR #1696³⁶³⁸ - Modified Scan Partitioner for Algorithms
- PR #1695³⁶³⁹ - This thread executors
- PR #1694³⁶⁴⁰ - Fixed #1688: Add timer counters for tfunc_total and exec_total
- PR #1693³⁶⁴¹ - Fix #1691: is_executor template specification fails for inherited executors
- PR #1692³⁶⁴² - Fixed #1662: Possible exception source in coalescing_message_handler
- Issue #1691³⁶⁴³ - is_executor template specification fails for inherited executors
- PR #1690³⁶⁴⁴ - added macro for non-intrusive serialization of classes without a default c-tor
- PR #1689³⁶⁴⁵ - Replace value_or_error with custom storage, unify future_data state
- Issue #1688³⁶⁴⁶ - Add timer counters for tfunc_total and exec_total
- PR #1687³⁶⁴⁷ - Fixed interval timer
- PR #1686³⁶⁴⁸ - Fixing cmake warnings about not existing pseudo target dependencies
- PR #1685³⁶⁴⁹ - Converting partitioners to use bulk async execute
- PR #1683³⁶⁵⁰ - Adds a tool for inspect that checks for character limits
- PR #1682³⁶⁵¹ - Change project name to (uppercase) HPX
- PR #1681³⁶⁵² - Counter shortnames

³⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1704>
³⁶³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1703>
³⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/1702>
³⁶³³ <https://github.com/STELLAR-GROUP/hpx/pull/1701>
³⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1700>
³⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1699>
³⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1698>
³⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1697>
³⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1696>
³⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1695>
³⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1694>
³⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1693>
³⁶⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1692>
³⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1691>
³⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1690>
³⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1689>
³⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1688>
³⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1687>
³⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1686>
³⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1685>
³⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1683>
³⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1682>
³⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1681>

- PR #1680³⁶⁵³ - Extended Non-intrusive Serialization to Ease Usage for Library Developers
- PR #1679³⁶⁵⁴ - Working on 1544: More executor changes
- PR #1678³⁶⁵⁵ - Transpose fixes
- PR #1677³⁶⁵⁶ - Improve Boost compatibility check
- PR #1676³⁶⁵⁷ - 1d stencil fix
- Issue #1675³⁶⁵⁸ - hpx project name is not HPX
- PR #1674³⁶⁵⁹ - Fixing the MPI parcelport
- PR #1673³⁶⁶⁰ - added move semantics to map/vector deserialization
- PR #1672³⁶⁶¹ - Vs2015 await
- PR #1671³⁶⁶² - Adapt transform for #1668
- PR #1670³⁶⁶³ - Started to work on #1668
- PR #1669³⁶⁶⁴ - Add this_thread_executors
- Issue #1667³⁶⁶⁵ - Apple build instructions in docs are out of date
- PR #1666³⁶⁶⁶ - Apex integration
- PR #1665³⁶⁶⁷ - Fixes an error with the whitespace check that showed the incorrect location of the error
- Issue #1664³⁶⁶⁸ - Inspect tool found incorrect endline whitespace
- PR #1663³⁶⁶⁹ - Improve use of locks
- Issue #1662³⁶⁷⁰ - Possible exception source in coalescing_message_handler
- PR #1661³⁶⁷¹ - Added support for 128bit number serialization
- PR #1660³⁶⁷² - Serialization 128bits
- PR #1659³⁶⁷³ - Implemented inner_product and adjacent_diff algos
- PR #1658³⁶⁷⁴ - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1657³⁶⁷⁵ - Use of shared_ptr in io_service_pool changed to unique_ptr

³⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1680>

³⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1679>

³⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1678>

³⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1677>

³⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1676>

³⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1675>

³⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1674>

³⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1673>

³⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1672>

³⁶⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1671>

³⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1670>

³⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1669>

³⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1667>

³⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1666>

³⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1665>

³⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1664>

³⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1663>

³⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1662>

³⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1661>

³⁶⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1660>

³⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1659>

³⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1658>

³⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1657>

- Issue #1656³⁶⁷⁶ - 1d_stencil codes all have wrong factor
- PR #1654³⁶⁷⁷ - When using runtime_mode_connect, find the correct localhost public ip address
- PR #1653³⁶⁷⁸ - Fixing 1617
- PR #1652³⁶⁷⁹ - Remove traits::action_may_require_id_splitting
- PR #1651³⁶⁸⁰ - Fixed performance counters related to AGAS cache timings
- PR #1650³⁶⁸¹ - Remove leftovers of traits::type_size
- PR #1649³⁶⁸² - Shorten target names on Windows to shorten used path names
- PR #1648³⁶⁸³ - Fixing problems introduced by merging #1623 for older compilers
- PR #1647³⁶⁸⁴ - Simplify running automatic builds on Windows
- Issue #1646³⁶⁸⁵ - Cache insert and update performance counters are broken
- Issue #1644³⁶⁸⁶ - Remove leftovers of traits::type_size
- Issue #1643³⁶⁸⁷ - Remove traits::action_may_require_id_splitting
- PR #1642³⁶⁸⁸ - Adds spell checker to the inspect tool for qbk and doxygen comments
- PR #1640³⁶⁸⁹ - First step towards fixing 688
- PR #1639³⁶⁹⁰ - Re-apply remaining changes from limit_dataflow_recursion branch
- PR #1638³⁶⁹¹ - This fixes possible deadlock in the test ignore_while_locked_1485
- PR #1637³⁶⁹² - Fixing hpx::wait_all() invoked with two vector<future<T>>
- PR #1636³⁶⁹³ - Partially re-apply changes from limit_dataflow_recursion branch
- PR #1635³⁶⁹⁴ - Adding missing test for #1572
- PR #1634³⁶⁹⁵ - Revert “Limit recursion-depth in dataflow to a configurable constant”
- PR #1633³⁶⁹⁶ - Add command line option to ignore batch environment
- PR #1631³⁶⁹⁷ - hpx::lcos::queue exhibits strange behavior
- PR #1630³⁶⁹⁸ - Fixed endlne_whitespace_check.cpp to detect lines with only whitespace

³⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1656>

³⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1654>

³⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1653>

³⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1652>

³⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1651>

³⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1650>

³⁶⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1649>

³⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1648>

³⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1647>

³⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1646>

³⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1644>

³⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1643>

³⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1642>

³⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1640>

³⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1639>

³⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1638>

³⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1637>

³⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1636>

³⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1635>

³⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1634>

³⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1633>

³⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1631>

³⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1630>

- Issue #1629³⁶⁹⁹ - Inspect trailing whitespace checker problem
- PR #1628³⁷⁰⁰ - Removed meaningless const qualifiers. Minor icpc fix.
- PR #1627³⁷⁰¹ - Fixing the queue LCO and add example demonstrating its use
- PR #1626³⁷⁰² - Deprecating get_gid(), add get_id() and get_unmanaged_id()
- PR #1625³⁷⁰³ - Allowing to specify whether to send credits along with message
- Issue #1624³⁷⁰⁴ - Lifetime issue
- Issue #1623³⁷⁰⁵ - hpx::wait_all() invoked with two vector<future<T>> fails
- PR #1622³⁷⁰⁶ - Executor partitioners
- PR #1621³⁷⁰⁷ - Clean up coroutines implementation
- Issue #1620³⁷⁰⁸ - Revert #1535
- PR #1619³⁷⁰⁹ - Fix result type calculation for hpx::make_continuation
- PR #1618³⁷¹⁰ - Fixing RDTSC on Xeon/Phi
- Issue #1617³⁷¹¹ - hpx cmake not working when run as a subproject
- Issue #1616³⁷¹² - cmake problem resulting in RDTSC not working correctly for Xeon Phi creates very strange results for duration counters
- Issue #1615³⁷¹³ - hpx::make_continuation requires input and output to be the same
- PR #1614³⁷¹⁴ - Fixed remove copy test
- Issue #1613³⁷¹⁵ - Dataflow causes stack overflow
- PR #1612³⁷¹⁶ - Modified foreach partitioner to use bulk execute
- PR #1611³⁷¹⁷ - Limit recursion-depth in dataflow to a configurable constant
- PR #1610³⁷¹⁸ - Increase timeout for CircleCI
- PR #1609³⁷¹⁹ - Refactoring thread manager, mainly extracting thread pool
- PR #1608³⁷²⁰ - Fixed running multiple localities without localities parameter
- PR #1607³⁷²¹ - More algorithm fixes to adjacentfind

³⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1629>

³⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1628>

³⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1627>

³⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1626>

³⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1625>

³⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1624>

³⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1623>

³⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1622>

³⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1621>

³⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1620>

³⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1619>

³⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1618>

³⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1617>

³⁷¹² <https://github.com/STELLAR-GROUP/hpx/issues/1616>

³⁷¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1615>

³⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1614>

³⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1613>

³⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1612>

³⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1611>

³⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1610>

³⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1609>

³⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1608>

³⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1607>

- Issue #1606³⁷²² - Running without localities parameter binds to bogus port range
- Issue #1605³⁷²³ - Too many serializations
- PR #1604³⁷²⁴ - Changes the HPX image into a hyperlink
- PR #1601³⁷²⁵ - Fixing problems with remove_copy algorithm tests
- PR #1600³⁷²⁶ - Actions with ids cleanup
- PR #1599³⁷²⁷ - Duplicate binding of global ids should fail
- PR #1598³⁷²⁸ - Fixing array access
- PR #1597³⁷²⁹ - Improved the reliability of connecting/disconnecting localities
- Issue #1596³⁷³⁰ - Duplicate id binding should fail
- PR #1595³⁷³¹ - Fixing more cmake config constants
- PR #1594³⁷³² - Fixing preprocessor constant used to enable C++11 chrono
- PR #1593³⁷³³ - Adding operator() for hpx::launch
- Issue #1592³⁷³⁴ - Error (typo) in the docs
- Issue #1590³⁷³⁵ - CMake fails when CMAKE_BINARY_DIR contains '+'.
- Issue #1589³⁷³⁶ - Disconnecting a locality results in segfault using heartbeat example
- PR #1588³⁷³⁷ - Fix doc string for config option HPX_WITH_EXAMPLES
- PR #1586³⁷³⁸ - Fixing 1493
- PR #1585³⁷³⁹ - Additional Check for Inspect Tool to detect Endline Whitespace
- Issue #1584³⁷⁴⁰ - Clean up coroutines implementation
- PR #1583³⁷⁴¹ - Adding a check for end line whitespace
- PR #1582³⁷⁴² - Attempt to fix assert firing after scheduling loop was exited
- PR #1581³⁷⁴³ - Fixed adjacentfind_binary test
- PR #1580³⁷⁴⁴ - Prevent some of the internal cmake lists from growing indefinitely

³⁷²² <https://github.com/STELLAR-GROUP/hpx/issues/1606>

³⁷²³ <https://github.com/STELLAR-GROUP/hpx/issues/1605>

³⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1604>

³⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1601>

³⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1600>

³⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1599>

³⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1598>

³⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1597>

³⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1596>

³⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1595>

³⁷³² <https://github.com/STELLAR-GROUP/hpx/pull/1594>

³⁷³³ <https://github.com/STELLAR-GROUP/hpx/pull/1593>

³⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1592>

³⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1590>

³⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1589>

³⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1588>

³⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1586>

³⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1585>

³⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1584>

³⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1583>

³⁷⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1582>

³⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1581>

³⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1580>

- PR #1579³⁷⁴⁵ - Removing type_size trait, replacing it with special archive type
- Issue #1578³⁷⁴⁶ - Remove demangle_helper
- PR #1577³⁷⁴⁷ - Get ptr problems
- Issue #1576³⁷⁴⁸ - Refactor async, dataflow, and future::then
- PR #1575³⁷⁴⁹ - Fixing tests for parallel rotate
- PR #1574³⁷⁵⁰ - Cleaning up schedulers
- PR #1573³⁷⁵¹ - Fixing thread pool executor
- PR #1572³⁷⁵² - Fixing number of configured localities
- PR #1571³⁷⁵³ - Reimplement decay
- PR #1570³⁷⁵⁴ - Refactoring async, apply, and dataflow APIs
- PR #1569³⁷⁵⁵ - Changed range for mach-o library lookup
- PR #1568³⁷⁵⁶ - Mark decltype support as required
- PR #1567³⁷⁵⁷ - Removed const from algorithms
- Issue #1566³⁷⁵⁸ - CMAKE Configuration Test Failures for clang 3.5 on debian
- PR #1565³⁷⁵⁹ - Dylib support
- PR #1564³⁷⁶⁰ - Converted partitioners and some algorithms to use executors
- PR #1563³⁷⁶¹ - Fix several #includes for Boost.Preprocessor
- PR #1562³⁷⁶² - Adding configuration option disabling/enabling all message handlers
- PR #1561³⁷⁶³ - Removed all occurrences of boost::move replacing it with std::move
- Issue #1560³⁷⁶⁴ - Leftover HPX_REGISTER_ACTION_DECLARATION_2
- PR #1558³⁷⁶⁵ - Revisit async/apply SFINAE conditions
- PR #1557³⁷⁶⁶ - Removing type_size trait, replacing it with special archive type
- PR #1556³⁷⁶⁷ - Executor algorithms

³⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1579>

³⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1578>

³⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1577>

³⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1576>

³⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1575>

³⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1574>

³⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1573>

³⁷⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1572>

³⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1571>

³⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1570>

³⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1569>

³⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1568>

³⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1567>

³⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1566>

³⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1565>

³⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1564>

³⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1563>

³⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1562>

³⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1561>

³⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1560>

³⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1558>

³⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1557>

³⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1556>

- PR #1555³⁷⁶⁸ - Remove the necessity to specify archive flags on the receiving end
- PR #1554³⁷⁶⁹ - Removing obsolete Boost.Serialization macros
- PR #1553³⁷⁷⁰ - Properly fix HPX_DEFINE_*_ACTION macros
- PR #1552³⁷⁷¹ - Fixed algorithms relying on copy_if implementation
- PR #1551³⁷⁷² - Pxfs - Modifying FindOrangeFS.cmake based on OrangeFS 2.9.X
- Issue #1550³⁷⁷³ - Passing plain identifier inside HPX_DEFINE_PLAIN_ACTION_1
- PR #1549³⁷⁷⁴ - Fixing intel14/libstdc++4.4
- PR #1548³⁷⁷⁵ - Moving raw_ptr to detail namespace
- PR #1547³⁷⁷⁶ - Adding support for executors to future.then
- PR #1546³⁷⁷⁷ - Executor traits result types
- PR #1545³⁷⁷⁸ - Integrate executors with dataflow
- PR #1543³⁷⁷⁹ - Fix potential zero-copy for primarynamespace::bulk_service_async et.al.
- PR #1542³⁷⁸⁰ - Merging HPX0.9.10 into pxfs branch
- PR #1541³⁷⁸¹ - Removed stale cmake tests, unused since the great cmake refactoring
- PR #1540³⁷⁸² - Fix idle-rate on platforms without TSC
- PR #1539³⁷⁸³ - Reporting situation if zero-copy-serialization was performed by a parcel generated from a plain apply/async
- PR #1538³⁷⁸⁴ - Changed return type of bulk executors and added test
- Issue #1537³⁷⁸⁵ - Incorrect cpuid config tests
- PR #1536³⁷⁸⁶ - Changed return type of bulk executors and added test
- PR #1535³⁷⁸⁷ - Make sure promise::get_gid() can be called more than once
- PR #1534³⁷⁸⁸ - Fixed async_callback with bound callback
- PR #1533³⁷⁸⁹ - Updated the link in the documentation to a publically- accessible URL
- PR #1532³⁷⁹⁰ - Make sure sync primitives are not copyable nor movable

³⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1555>

³⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1554>

³⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1553>

³⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1552>

³⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1551>

³⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1550>

³⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1549>

³⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1548>

³⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1547>

³⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1546>

³⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1545>

³⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1543>

³⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1542>

³⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1541>

³⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1540>

³⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1539>

³⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1538>

³⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1537>

³⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1536>

³⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1535>

³⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1534>

³⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1533>

³⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1532>

- PR #1531³⁷⁹¹ - Fix unwrapped issue with future ranges of void type
- PR #1530³⁷⁹² - Serialization complex
- Issue #1528³⁷⁹³ - Unwrapped issue with future<void>
- Issue #1527³⁷⁹⁴ - HPX does not build with Boost 1.58.0
- PR #1526³⁷⁹⁵ - Added support for boost.multi_array serialization
- PR #1525³⁷⁹⁶ - Properly handle deferred futures, fixes #1506
- PR #1524³⁷⁹⁷ - Making sure invalid action argument types generate clear error message
- Issue #1522³⁷⁹⁸ - Need serialization support for boost multi array
- Issue #1521³⁷⁹⁹ - Remote async and zero-copy serialization optimizations don't play well together
- PR #1520³⁸⁰⁰ - Fixing UB whil registering polymorphic classes for serialization
- PR #1519³⁸⁰¹ - Making detail::condition_variable safe to use
- PR #1518³⁸⁰² - Fix when_some bug missing indices in its result
- Issue #1517³⁸⁰³ - Typo may affect CMake build system tests
- PR #1516³⁸⁰⁴ - Fixing Posix context
- PR #1515³⁸⁰⁵ - Fixing Posix context
- PR #1514³⁸⁰⁶ - Correct problems with loading dynamic components
- PR #1513³⁸⁰⁷ - Fixing intel glibc4 4
- Issue #1508³⁸⁰⁸ - memory and papi counters do not work
- Issue #1507³⁸⁰⁹ - Unrecognized Command Line Option Error causing exit status 0
- Issue #1506³⁸¹⁰ - Properly handle deferred futures
- PR #1505³⁸¹¹ - Adding #include - would not compile without this
- Issue #1502³⁸¹² - boost::filesystem::exists throws unexpected exception
- Issue #1501³⁸¹³ - hwloc configuration options are wrong for MIC

³⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1531>

³⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1530>

³⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1528>

³⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1527>

³⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1526>

³⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1525>

³⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1524>

³⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1522>

³⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1521>

³⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1520>

³⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1519>

³⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1518>

³⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1517>

³⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1516>

³⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1515>

³⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1514>

³⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1513>

³⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1508>

³⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1507>

³⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1506>

³⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1505>

³⁸¹² <https://github.com/STELLAR-GROUP/hpx/issues/1502>

³⁸¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1501>

- PR #1504³⁸¹⁴ - Making sure boost::filesystem::exists() does not throw
- PR #1500³⁸¹⁵ - Exit application on --hpx:version/-v and --hpx:info
- PR #1498³⁸¹⁶ - Extended task block
- PR #1497³⁸¹⁷ - Unique ptr serialization
- PR #1496³⁸¹⁸ - Unique ptr serialization (closed)
- PR #1495³⁸¹⁹ - Switching circleci build type to debug
- Issue #1494³⁸²⁰ - --hpx:version/-v does not exit after printing version information
- Issue #1493³⁸²¹ - add an hpx_ prefix to libraries and components to avoid name conflicts
- Issue #1492³⁸²² - Define and ensure limitations for arguments to async/apply
- PR #1489³⁸²³ - Enable idle rate counter on demand
- PR #1488³⁸²⁴ - Made sure detail::condition_variable can be safely destroyed
- PR #1487³⁸²⁵ - Introduced default (main) template implementation for ignore_while_checking
- PR #1486³⁸²⁶ - Add HPX inspect tool
- Issue #1485³⁸²⁷ - ignore_while_locked doesn't support all Lockable types
- PR #1484³⁸²⁸ - Docker image generation
- PR #1483³⁸²⁹ - Move external endian library into HPX
- PR #1482³⁸³⁰ - Actions with integer type ids
- Issue #1481³⁸³¹ - Sync primitives safe destruction
- Issue #1480³⁸³² - Move external/boost/endian into hpx/util
- Issue #1478³⁸³³ - Boost inspect violations
- PR #1479³⁸³⁴ - Adds serialization for arrays; some further/minor fixes
- PR #1477³⁸³⁵ - Fixing problems with the Intel compiler using a GCC 4.4 std library
- PR #1476³⁸³⁶ - Adding hpx::lcos::latch and hpx::lcos::local::latch

³⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1504>

³⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1500>

³⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1498>

³⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1497>

³⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1496>

³⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1495>

³⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1494>

³⁸²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1493>

³⁸²² <https://github.com/STELLAR-GROUP/hpx/issues/1492>

³⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/1489>

3824 <https://github.com/STELLAR-GROUP/hpx/pull/1488>3825 <https://github.com/STELLAR-GROUP/hpx/pull/1487>3826 <https://github.com/STELLAR-GROUP/hpx/pull/1486>3827 <https://github.com/STELLAR-GROUP/hpx/issues/1485>

3828 <https://github.com/STELLAR-GROUP/hpx/pull/1484>

3829 <https://github.com/STELLAR-GROUP/hpx/pull/1483>

3830 <https://github.com/STELLAR-GROUP/hpx/pull/1482>

3831 <https://github.com/STELLAR-GROUP/hpx/issues/1481>

3832 <https://github.com/STELLAR-GROUP/hpx/issues/1480>

3833 <https://github.com/STELLAR-GROUP/hpx/issues/1478>

3834 <https://github.com/STELLAR-GROUP/hpx/pull/1479>

3835 <https://github.com/STELLAR-GROUP/hpx/pull/1477>

3836 <https://github.com/STELLAR-GROUP/hpx/pull/1476>

2.10. Releases

1821

- Issue #1475³⁸³⁷ - Boost inspect violations
- PR #1473³⁸³⁸ - Fixing action move tests
- Issue #1471³⁸³⁹ - Sync primitives should not be movable
- PR #1470³⁸⁴⁰ - Removing `hpx::util::polymorphic_factory`
- PR #1468³⁸⁴¹ - Fixed container creation
- Issue #1467³⁸⁴² - HPX application fail during finalization
- Issue #1466³⁸⁴³ - HPX doesn't pick up Torque's nodefile on SuperMIC
- Issue #1464³⁸⁴⁴ - HPX option for pre and post bootstrap performance counters
- PR #1463³⁸⁴⁵ - Replacing `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1462³⁸⁴⁶ - Consolidated task_region with N4411
- PR #1461³⁸⁴⁷ - Consolidate inconsistent CMake option names
- Issue #1460³⁸⁴⁸ - Which malloc is actually used? or at least which one is HPX built with
- Issue #1459³⁸⁴⁹ - Make cmake configure step fail explicitly if compiler version is not supported
- Issue #1458³⁸⁵⁰ - Update `parallel::task_region` with N4411
- PR #1456³⁸⁵¹ - Consolidating `new_<>()`
- Issue #1455³⁸⁵² - Replace `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1454³⁸⁵³ - Removed harmful std::moves from return statements
- PR #1453³⁸⁵⁴ - Use range-based for-loop instead of Boost.Foreach
- PR #1452³⁸⁵⁵ - C++ feature tests
- PR #1451³⁸⁵⁶ - When serializing, pass archive flags to traits::get_type_size
- Issue #1450³⁸⁵⁷ - traits::get_type_size needs archive flags to enable zero_copy optimizations
- Issue #1449³⁸⁵⁸ - “couldn't create performance counter” - AGAS
- Issue #1448³⁸⁵⁹ - Replace distributing factories with `new_<T[]>(...)`

³⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1475>

³⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1473>

³⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1471>

³⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1470>

³⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1468>

³⁸⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1467>

³⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1466>

³⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1464>

³⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1463>

³⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1462>

³⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1461>

³⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1460>

³⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1459>

³⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1458>

³⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1456>

³⁸⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1455>

³⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1454>

³⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1453>

³⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1452>

³⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1451>

³⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1450>

³⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1449>

³⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1448>

- PR #1447³⁸⁶⁰ - Removing obsolete remote_object component
- PR #1446³⁸⁶¹ - Hpx serialization
- PR #1445³⁸⁶² - Replacing travis with circleci
- PR #1443³⁸⁶³ - Always stripping HPX command line arguments before executing start function
- PR #1442³⁸⁶⁴ - Adding –hpx:bind=none to disable thread affinities
- Issue #1439³⁸⁶⁵ - Libraries get linked in multiple times, RPATH is not properly set
- PR #1438³⁸⁶⁶ - Removed superfluous typedefs
- Issue #1437³⁸⁶⁷ - hpx::init() should strip HPX-related flags from argv
- Issue #1436³⁸⁶⁸ - Add strong scaling option to htts
- PR #1435³⁸⁶⁹ - Adding async_cb, async_continue_cb, and async_colocated_cb
- PR #1434³⁸⁷⁰ - Added missing install rule, removed some dead CMake code
- PR #1433³⁸⁷¹ - Add GitExternal and SubProject cmake scripts from eyescale/cmake repo
- Issue #1432³⁸⁷² - Add command line flag to disable thread pinning
- PR #1431³⁸⁷³ - Fix #1423
- Issue #1430³⁸⁷⁴ - Inconsistent CMake option names
- Issue #1429³⁸⁷⁵ - Configure setting HPX_HAVE_PARCELPORT_MPI is ignored
- PR #1428³⁸⁷⁶ - Fixes #1419 (closed)
- PR #1427³⁸⁷⁷ - Adding stencil_iterator and transform_iterator
- PR #1426³⁸⁷⁸ - Fixes #1419
- PR #1425³⁸⁷⁹ - During serialization memory allocation should honour allocator chunk size
- Issue #1424³⁸⁸⁰ - chunk allocation during serialization does not use memory pool/allocator chunk size
- Issue #1423³⁸⁸¹ - Remove HPX_STD_UNIQUE_PTR
- Issue #1422³⁸⁸² - hpx:threads=all allocates too many os threads

³⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1447>

³⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1446>

³⁸⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1445>

³⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1443>

³⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1442>

³⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1439>

³⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1438>

³⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1437>

³⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1436>

³⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1435>

³⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1434>

³⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1433>

³⁸⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1432>

³⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1431>

³⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1430>

³⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1429>

³⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1428>

³⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1427>

³⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1426>

³⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1425>

³⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1424>

³⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1423>

³⁸⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1422>

- PR #1420³⁸⁸³ - added .travis.yml
- Issue #1419³⁸⁸⁴ - Unify enums: `hpx::runtime::state` and `hpx::state`
- PR #1416³⁸⁸⁵ - Adding travis builder
- Issue #1414³⁸⁸⁶ - Correct directory for dispatch_gcc46.hpp iteration
- Issue #1410³⁸⁸⁷ - Set operation algorithms
- Issue #1389³⁸⁸⁸ - Parallel algorithms relying on scan partitioner break for small number of elements
- Issue #1325³⁸⁸⁹ - Exceptions thrown during parcel handling are not handled correctly
- Issue #1315³⁸⁹⁰ - Errors while running performance tests
- Issue #1309³⁸⁹¹ - `hpx::vector` partitions are not easily extendable by applications
- PR #1300³⁸⁹² - Added serialization/de-serialization to examples.tuplespace
- Issue #1251³⁸⁹³ - `hpx::threads::get_thread_count` doesn't consider pending threads
- Issue #1008³⁸⁹⁴ - Decrease in application performance overtime; occasional spikes of major slowdown
- Issue #1001³⁸⁹⁵ - Zero copy serialization raises assert
- Issue #721³⁸⁹⁶ - Make HPX usable for Xeon Phi
- Issue #524³⁸⁹⁷ - Extend scheduler to support threads which can't be stolen

2.10.15 HPX V0.9.10 (Mar 24, 2015)

General changes

This is the 12th official release of *HPX*. It coincides with the 7th anniversary of the first commit to our source code repository. Since then, we have seen over 12300 commits amounting to more than 220000 lines of C++ code.

The major focus of this release was to improve the reliability of large scale runs. We believe to have achieved this goal as we now can reliably run *HPX* applications on up to ~24k cores. We have also shown that *HPX* can be used with success for symmetric runs (applications using both, host cores and Intel Xeon/Phi coprocessors). This is a huge step forward in terms of the usability of *HPX*. The main focus of this work involved isolating the causes of the segmentation faults at start up and shut down. Many of these issues were discovered to be the result of the suspension of threads which hold locks.

A very important improvement introduced with this release is the refactoring of the code representing our parcel-port implementation. Parcel- ports can now be implemented by 3rd parties as independent plugins which are dynamically loaded at runtime (static linking of parcel-ports is also supported). This refactoring also includes a massive improvement of the performance of our existing parcel-ports. We were able to significantly reduce the networking latencies

³⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1420>

³⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1419>

³⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1416>

³⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1414>

3887 <https://github.com/STELLAR-GROUP/hpx/issues/1410>3888 <https://github.com/STELLAR-GROUP/hpx/issues/1389>3889 <https://github.com/STELLAR-GROUP/hpx/issues/1325>3890 <https://github.com/STELLAR-GROUP/hpx/issues/1315>3891 <https://github.com/STELLAR-GROUP/hpx/issues/1309>3892 <https://github.com/STELLAR-GROUP/hpx/pull/1300>3893 <https://github.com/STELLAR-GROUP/hpx/issues/1251>3894 <https://github.com/STELLAR-GROUP/hpx/issues/1008>3895 <https://github.com/STELLAR-GROUP/hpx/issues/1001>3896 <https://github.com/STELLAR-GROUP/hpx/issues/721>3897 <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 #839³⁸⁹⁸](#)).

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 #1338³⁸⁹⁹](#) for a documentation of our progress).

Breaking changes

With this release we put a lot of effort into changing the code base to be more compatible to C++11. These changes have caused the following issues for backward compatibility:

- Move to Variadics- All of the API now uses variadic templates. However, this change required to modify the argument sequence for some of the exiting API functions (`hpx::async_continue`, `hpx::apply_continue`, `hpx::when_each`, `hpx::wait_each`, synchronous invocation of actions).
- Changes to Macros- We also removed the macros `HPX_STD_FUNCTION` and `HPX_STD_TUPLE`. This shouldn't affect any user code as we replaced `HPX_STD_FUNCTION` with `hpx::util::function_nonser` which was the default expansion used for this macro. All `HPX` API functions which expect a `hpx::util::function_nonser` (or a `hpx::util::unique_function_nonser`) can now be transparently called with a compatible `std::function` instead. Similarly, `HPX_STD_TUPLE` was replaced by its default expansion as well: `hpx::util::tuple`.
- Changes to `hpx::unique_future`- `hpx::unique_future`, which was deprecated in the previous release for `hpx::future` is now completely removed from `HPX`. This completes the transition to a completely standards conforming implementation of `hpx::future`.
- Changes to Supported Compilers. Finally, in order to utilize more C++11 semantics, we have officially dropped support for GCC 4.4 and MSVC 2012. Please see our *Prerequisites* page for more details.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- [Issue #1402³⁹⁰⁰](#) - Internal shared_future serialization copies
- [Issue #1399³⁹⁰¹](#) - Build takes unusually long time...
- [Issue #1398³⁹⁰²](#) - Tests using the scan partitioner are broken on at least gcc 4.7 and intel compiler
- [Issue #1397³⁹⁰³](#) - Completely remove `hpx::unique_future`
- [Issue #1396³⁹⁰⁴](#) - Parallel scan algorithms with different initial values

³⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/839>

³⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1338>

³⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1402>

³⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1399>

³⁹⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1398>

³⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1397>

³⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1396>

- Issue #1395³⁹⁰⁵ - Race Condition - 1d_stencil_8 - SuperMIC
- Issue #1394³⁹⁰⁶ - “suspending thread while at least one lock is being held” - 1d_stencil_8 - SuperMIC
- Issue #1393³⁹⁰⁷ - SEGFAULT in 1d_stencil_8 on SuperMIC
- Issue #1392³⁹⁰⁸ - Fixing #1168
- Issue #1391³⁹⁰⁹ - Parallel Algorithms for scan partitioner for small number of elements
- Issue #1387³⁹¹⁰ - Failure with more than 4 localities
- Issue #1386³⁹¹¹ - Dispatching unhandled exceptions to outer user code
- Issue #1385³⁹¹² - Adding Copy algorithms, fixing parallel::copy_if
- Issue #1384³⁹¹³ - Fixing 1325
- Issue #1383³⁹¹⁴ - Fixed #504: Refactor Dataflow LCO to work with futures, this removes the dataflow component as it is obsolete
- Issue #1382³⁹¹⁵ - is_sorted, is_sorted_until and is_partitioned algorithms
- Issue #1381³⁹¹⁶ - fix for CMake versions prior to 3.1
- Issue #1380³⁹¹⁷ - resolved warning in CMake 3.1 and newer
- Issue #1379³⁹¹⁸ - Compilation error with papi
- Issue #1378³⁹¹⁹ - Towards safer migration
- Issue #1377³⁹²⁰ - HPXConfig.cmake should include TCMALLOC_LIBRARY and TCMALLOC_INCLUDE_DIR
- Issue #1376³⁹²¹ - Warning on uninitialized member
- Issue #1375³⁹²² - Fixing 1163
- Issue #1374³⁹²³ - Fixing the MSVC 12 release builder
- Issue #1373³⁹²⁴ - Modifying parallel search algorithm for zero length searches
- Issue #1372³⁹²⁵ - Modifying parallel search algorithm for zero length searches
- Issue #1371³⁹²⁶ - Avoid holding a lock during agas::inref while doing a credit split
- Issue #1370³⁹²⁷ - --hpx:bind throws unexpected error

³⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1395>

³⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1394>

³⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1393>

³⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1392>

³⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1391>

³⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1387>

³⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1386>

³⁹¹² <https://github.com/STELLAR-GROUP/hpx/issues/1385>

³⁹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1384>

³⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1383>

³⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1382>

³⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1381>

³⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1380>

³⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1379>

³⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1378>

³⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1377>

³⁹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1376>

³⁹²² <https://github.com/STELLAR-GROUP/hpx/issues/1375>

³⁹²³ <https://github.com/STELLAR-GROUP/hpx/issues/1374>

³⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1373>

³⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1372>

³⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1371>

³⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1370>

- Issue #1369³⁹²⁸ - Getting rid of (void) in loops
- Issue #1368³⁹²⁹ - Variadic templates support for tuple
- Issue #1367³⁹³⁰ - One last batch of variadic templates support
- Issue #1366³⁹³¹ - Fixing symbolic namespace hang
- Issue #1365³⁹³² - More held locks
- Issue #1364³⁹³³ - Add counters 1363
- Issue #1363³⁹³⁴ - Add thread overhead counters
- Issue #1362³⁹³⁵ - Std config removal
- Issue #1361³⁹³⁶ - Parcelport plugins
- Issue #1360³⁹³⁷ - Detuplify transfer_action
- Issue #1359³⁹³⁸ - Removed obsolete checks
- Issue #1358³⁹³⁹ - Fixing 1352
- Issue #1357³⁹⁴⁰ - Variadic templates support for runtime_support and components
- Issue #1356³⁹⁴¹ - fixed coordinate test for intel13
- Issue #1355³⁹⁴² - fixed coordinate.hpp
- Issue #1354³⁹⁴³ - Lexicographical Compare completed
- Issue #1353³⁹⁴⁴ - HPX should set Boost_ADDITIONAL VERSIONS flags
- Issue #1352³⁹⁴⁵ - Error: Cannot find action “ in type registry: HPX(bad_action_code)
- Issue #1351³⁹⁴⁶ - Variadic templates support for applicrs
- Issue #1350³⁹⁴⁷ - Actions simplification
- Issue #1349³⁹⁴⁸ - Variadic when and wait functions
- Issue #1348³⁹⁴⁹ - Added hpx_init header to test files
- Issue #1347³⁹⁵⁰ - Another batch of variadic templates support

³⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1369>

³⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1368>

³⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1367>

³⁹³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1366>

³⁹³² <https://github.com/STELLAR-GROUP/hpx/issues/1365>

³⁹³³ <https://github.com/STELLAR-GROUP/hpx/issues/1364>

³⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1363>

³⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1362>

³⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1361>

³⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1360>

³⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1359>

³⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1358>

³⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1357>

³⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1356>

³⁹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1355>

³⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1354>

³⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1353>

³⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1352>

³⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1351>

³⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1350>

³⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1349>

³⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1348>

³⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1347>

- Issue #1346³⁹⁵¹ - Segmented copy
- Issue #1345³⁹⁵² - Attempting to fix hangs during shutdown
- Issue #1344³⁹⁵³ - Std config removal
- Issue #1343³⁹⁵⁴ - Removing various distribution policies for hpx::vector
- Issue #1342³⁹⁵⁵ - Inclusive scan
- Issue #1341³⁹⁵⁶ - Exclusive scan
- Issue #1340³⁹⁵⁷ - Adding parallel::count for distributed data structures, adding tests
- Issue #1339³⁹⁵⁸ - Update argument order for transform_reduce
- Issue #1337³⁹⁵⁹ - Fix dataflow to handle properly ranges of futures
- Issue #1336³⁹⁶⁰ - dataflow needs to hold onto futures passed to it
- Issue #1335³⁹⁶¹ - Fails to compile with msvc14
- Issue #1334³⁹⁶² - Examples build problem
- Issue #1333³⁹⁶³ - Distributed transform reduce
- Issue #1332³⁹⁶⁴ - Variadic templates support for actions
- Issue #1331³⁹⁶⁵ - Some ambiguous calls of map::erase have been prevented by adding additional check in locality constructor.
- Issue #1330³⁹⁶⁶ - Defining Plain Actions does not work as described in the documentation
- Issue #1329³⁹⁶⁷ - Distributed vector cleanup
- Issue #1328³⁹⁶⁸ - Sync docs and comments with code in hello_world example
- Issue #1327³⁹⁶⁹ - Typos in docs
- Issue #1326³⁹⁷⁰ - Documentation and code diverged in Fibonacci tutorial
- Issue #1325³⁹⁷¹ - Exceptions thrown during parcel handling are not handled correctly
- Issue #1324³⁹⁷² - fixed bandwidth calculation
- Issue #1323³⁹⁷³ - mmap() failed to allocate thread stack due to insufficient resources

³⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1346>

³⁹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1345>

³⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1344>

³⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1343>

³⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1342>

³⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1341>

³⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1340>

³⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1339>

³⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1337>

³⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1336>

³⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1335>

³⁹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1334>

³⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1333>

³⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1332>

³⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1331>

³⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1330>

³⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1329>

³⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1328>

³⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1327>

³⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1326>

³⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1325>

³⁹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1324>

³⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1323>

- Issue #1322³⁹⁷⁴ - HPX fails to build aa182cf
- Issue #1321³⁹⁷⁵ - Limiting size of outgoing messages while coalescing parcels
- Issue #1320³⁹⁷⁶ - passing a future with launch::deferred in remote function call causes hang
- Issue #1319³⁹⁷⁷ - An exception when tries to specify number high priority threads with abp-priority
- Issue #1318³⁹⁷⁸ - Unable to run program with abp-priority and numa-sensitivity enabled
- Issue #1317³⁹⁷⁹ - N4071 Search/Search_n finished, minor changes
- Issue #1316³⁹⁸⁰ - Add config option to make -Ihpx.run_hpx_main!=1 the default
- Issue #1314³⁹⁸¹ - Variadic support for async and apply
- Issue #1313³⁹⁸² - Adjust when_any/some to the latest proposed interfaces
- Issue #1312³⁹⁸³ - Fixing #857: hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #1311³⁹⁸⁴ - Distributed get'er/set'er_values for distributed vector
- Issue #1310³⁹⁸⁵ - Crashing in hpx::parcelset::policies::mpi::connection_handler::handle_messages() on Super-MIC
- Issue #1308³⁹⁸⁶ - Unable to execute an application with --hpx:threads
- Issue #1307³⁹⁸⁷ - merge_graph linking issue
- Issue #1306³⁹⁸⁸ - First batch of variadic templates support
- Issue #1305³⁹⁸⁹ - Create a compiler wrapper
- Issue #1304³⁹⁹⁰ - Provide a compiler wrapper for hpx
- Issue #1303³⁹⁹¹ - Drop support for GCC44
- Issue #1302³⁹⁹² - Fixing #1297
- Issue #1301³⁹⁹³ - Compilation error when tried to use boost range iterators with wait_all
- Issue #1298³⁹⁹⁴ - Distributed vector
- Issue #1297³⁹⁹⁵ - Unable to invoke component actions recursively

³⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1322>

³⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1321>

³⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1320>

³⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1319>

³⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1318>

³⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1317>

³⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1316>

³⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1314>

³⁹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1313>

³⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1312>

³⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1311>

³⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1310>

³⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1308>

³⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1307>

³⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1306>

³⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1305>

³⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1304>

³⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1303>

³⁹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1302>

³⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1301>

³⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1298>

³⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1297>

- Issue #1294³⁹⁹⁶ - HDF5 build error
- Issue #1275³⁹⁹⁷ - The parcelport implementation is non-optimal
- Issue #1267³⁹⁹⁸ - Added classes and unit tests for local_file, orangefs_file and pxfs_file
- Issue #1264³⁹⁹⁹ - Error “assertion ‘!m_fun’ failed” randomly occurs when using TCP
- Issue #1254⁴⁰⁰⁰ - thread binding seems to not work properly
- Issue #1220⁴⁰⁰¹ - parallel::copy_if is broken
- Issue #1217⁴⁰⁰² - Find a better way of fixing the issue patched by #1216
- Issue #1168⁴⁰⁰³ - Starting HPX on Cray machines using aprun isn’t working correctly
- Issue #1085⁴⁰⁰⁴ - Replace startup and shutdown barriers with broadcasts
- Issue #981⁴⁰⁰⁵ - With SLURM, –hpx:threads=8 should not be necessary
- Issue #857⁴⁰⁰⁶ - hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #850⁴⁰⁰⁷ - “flush” not documented
- Issue #763⁴⁰⁰⁸ - Create buildbot instance that uses std::bind as HPX_STD_BIND
- Issue #680⁴⁰⁰⁹ - Convert parcel ports into a plugin system
- Issue #582⁴⁰¹⁰ - Make exception thrown from HPX threads available from hpx::init
- Issue #504⁴⁰¹¹ - Refactor Dataflow LCO to work with futures
- Issue #196⁴⁰¹² - Don’t store copies of the locality network metadata in the gva table

2.10.16 HPX V0.9.9 (Oct 31, 2014, codename Spooky)

General changes

We have had over 1500 commits since the last release and we have closed over 200 tickets (bugs, feature requests, pull requests, etc.). These are by far the largest numbers of commits and resolved issues for any of the *HPX* releases so far. We are especially happy about the large number of people who contributed for the first time to *HPX*.

- We completed the transition from the older (non-conforming) implementation of hpx::future to the new and fully conforming version by removing the old code and by renaming the type hpx::unique_future to hpx::future. In order to maintain backwards compatibility with existing code which uses the type hpx::unique_future we support the configuration variable HPX_UNIQUE_FUTURE_ALIAS. If this variable is set to ON while running cmake it will additionally define a template alias for this type.

³⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1294>

³⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1275>

3998 <https://github.com/STELLAR-GROUP/hpx/issues/1267>3999 <https://github.com/STELLAR-GROUP/hpx/issues/1264>4000 <https://github.com/STELLAR-GROUP/hpx/issues/1254>4001 <https://github.com/STELLAR-GROUP/hpx/issues/1220>4002 <https://github.com/STELLAR-GROUP/hpx/issues/1217>4003 <https://github.com/STELLAR-GROUP/hpx/issues/1168>4004 <https://github.com/STELLAR-GROUP/hpx/issues/1085>4005 <https://github.com/STELLAR-GROUP/hpx/issues/981>4006 <https://github.com/STELLAR-GROUP/hpx/issues/857>4007 <https://github.com/STELLAR-GROUP/hpx/issues/850>4008 <https://github.com/STELLAR-GROUP/hpx/issues/763>4009 <https://github.com/STELLAR-GROUP/hpx/issues/680>4010 <https://github.com/STELLAR-GROUP/hpx/issues/582>4011 <https://github.com/STELLAR-GROUP/hpx/issues/504>4012 <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 #1204⁴⁰¹³](#).
- We started to implement various proposals to the C++ Standardization committee related to parallelism and concurrency, most notably [N4409⁴⁰¹⁴](#) (Working Draft, Technical Specification for C++ Extensions for Parallelism), [N4411⁴⁰¹⁵](#) (Task Region Rev. 3), and [N4313⁴⁰¹⁶](#) (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 Website⁴⁰¹⁷](#).

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue [#1296⁴⁰¹⁸](#) - Rename make_error_future to make_exceptional_future, adjust to N4123
- Issue [#1295⁴⁰¹⁹](#) - building issue
- Issue [#1293⁴⁰²⁰](#) - Transpose example
- Issue [#1292⁴⁰²¹](#) - Wrong abs() function used in example
- Issue [#1291⁴⁰²²](#) - non-synchronized shift operators have been removed
- Issue [#1290⁴⁰²³](#) - RDTSCP is defined as true for Xeon Phi build
- Issue [#1289⁴⁰²⁴](#) - Fixing 1288
- Issue [#1288⁴⁰²⁵](#) - Add new performance counters
- Issue [#1287⁴⁰²⁶](#) - Hierarchy scheduler broken performance counters

⁴⁰¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1204>

⁴⁰¹⁴ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf>

⁴⁰¹⁵ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

⁴⁰¹⁶ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

⁴⁰¹⁷ <http://rostam.cct.lsu.edu/>

⁴⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1296>

⁴⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1295>

⁴⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1293>

⁴⁰²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1292>

⁴⁰²² <https://github.com/STELLAR-GROUP/hpx/issues/1291>

⁴⁰²³ <https://github.com/STELLAR-GROUP/hpx/issues/1290>

⁴⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1289>

⁴⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1288>

⁴⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1287>

- Issue #1286⁴⁰²⁷ - Algorithm cleanup
- Issue #1285⁴⁰²⁸ - Broken Links in Documentation
- Issue #1284⁴⁰²⁹ - Uninitialized copy
- Issue #1283⁴⁰³⁰ - missing boost::scoped_ptr includes
- Issue #1282⁴⁰³¹ - Update documentation of build options for schedulers
- Issue #1281⁴⁰³² - reset idle rate counter
- Issue #1280⁴⁰³³ - Bug when executing on Intel MIC
- Issue #1279⁴⁰³⁴ - Add improved when_all/wait_all
- Issue #1278⁴⁰³⁵ - Implement improved when_all/wait_all
- Issue #1277⁴⁰³⁶ - feature request: get access to argc argv and variables_map
- Issue #1276⁴⁰³⁷ - Remove merging map
- Issue #1274⁴⁰³⁸ - Weird (wrong) string code in papi.cpp
- Issue #1273⁴⁰³⁹ - Sequential task execution policy
- Issue #1272⁴⁰⁴⁰ - Avoid CMake name clash for Boost.Thread library
- Issue #1271⁴⁰⁴¹ - Updates on HPX Test Units
- Issue #1270⁴⁰⁴² - hpx/util/safe_lexical_cast.hpp is added
- Issue #1269⁴⁰⁴³ - Added default value for “LIB” cmake variable
- Issue #1268⁴⁰⁴⁴ - Memory Counters not working
- Issue #1266⁴⁰⁴⁵ - FindHPX.cmake is not installed
- Issue #1263⁴⁰⁴⁶ - apply_remote test takes too long
- Issue #1262⁴⁰⁴⁷ - Chrono cleanup
- Issue #1261⁴⁰⁴⁸ - Need make install for papi counters and this builds all the examples
- Issue #1260⁴⁰⁴⁹ - Documentation of Stencil example claims

⁴⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1286>

⁴⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1285>

⁴⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1284>

⁴⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1283>

⁴⁰³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1282>

⁴⁰³² <https://github.com/STELLAR-GROUP/hpx/issues/1281>

⁴⁰³³ <https://github.com/STELLAR-GROUP/hpx/issues/1280>

⁴⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1279>

⁴⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1278>

⁴⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1277>

⁴⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1276>

⁴⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1274>

⁴⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1273>

⁴⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1272>

⁴⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1271>

⁴⁰⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1270>

⁴⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1269>

⁴⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1268>

⁴⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1266>

⁴⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1263>

⁴⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1262>

⁴⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1261>

⁴⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1260>

- Issue #1259⁴⁰⁵⁰ - Avoid double-linking Boost on Windows
- Issue #1257⁴⁰⁵¹ - Adding additional parameter to create_thread
- Issue #1256⁴⁰⁵² - added buildbot changes to release notes
- Issue #1255⁴⁰⁵³ - Cannot build MiniGhost
- Issue #1253⁴⁰⁵⁴ - hpx::thread defects
- Issue #1252⁴⁰⁵⁵ - HPX_PREFIX is too fragile
- Issue #1250⁴⁰⁵⁶ - switch_to_fiber_emulation does not work properly
- Issue #1249⁴⁰⁵⁷ - Documentation is generated under Release folder
- Issue #1248⁴⁰⁵⁸ - Fix usage of hpx_generic_coroutine_context and get tests passing on powerpc
- Issue #1247⁴⁰⁵⁹ - Dynamic linking error
- Issue #1246⁴⁰⁶⁰ - Make cpuid.cpp C++11 compliant
- Issue #1245⁴⁰⁶¹ - HPX fails on startup (setting thread affinity mask)
- Issue #1244⁴⁰⁶² - HPX_WITH_RDTSC configure test fails, but should succeed
- Issue #1243⁴⁰⁶³ - CTest dashboard info for CSCS CDash drop location
- Issue #1242⁴⁰⁶⁴ - Mac fixes
- Issue #1241⁴⁰⁶⁵ - Failure in Distributed with Boost 1.56
- Issue #1240⁴⁰⁶⁶ - fix a race condition in examples.diskperf
- Issue #1239⁴⁰⁶⁷ - fix wait_each in examples.diskperf
- Issue #1238⁴⁰⁶⁸ - Fixed #1237: hpx::util::portable_binary_iarchive failed
- Issue #1237⁴⁰⁶⁹ - hpx::util::portable_binary_iarchive faileds
- Issue #1235⁴⁰⁷⁰ - Fixing clang warnings and errors
- Issue #1234⁴⁰⁷¹ - TCP runs fail: Transport endpoint is not connected
- Issue #1233⁴⁰⁷² - Making sure the correct number of threads is registered with AGAS

⁴⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1259>

⁴⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1257>

⁴⁰⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1256>

⁴⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1255>

⁴⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1253>

⁴⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1252>

⁴⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1250>

⁴⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1249>

⁴⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1248>

⁴⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1247>

⁴⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1246>

⁴⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1245>

⁴⁰⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1244>

⁴⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1243>

⁴⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1242>

⁴⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1241>

⁴⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1240>

⁴⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1239>

⁴⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1238>

⁴⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1237>

⁴⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1235>

⁴⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1234>

⁴⁰⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1233>

- Issue #1232⁴⁰⁷³ - Fixing race in wait_xxx
- Issue #1231⁴⁰⁷⁴ - Parallel minmax
- Issue #1230⁴⁰⁷⁵ - Distributed run of 1d_stencil_8 uses less threads than spec. & sometimes gives errors
- Issue #1229⁴⁰⁷⁶ - Unstable number of threads
- Issue #1228⁴⁰⁷⁷ - HPX link error (cmake / MPI)
- Issue #1226⁴⁰⁷⁸ - Warning about struct/class thread_counters
- Issue #1225⁴⁰⁷⁹ - Adding parallel::replace etc
- Issue #1224⁴⁰⁸⁰ - Extending dataflow to pass through non-future arguments
- Issue #1223⁴⁰⁸¹ - Remaining find algorithms implemented, N4071
- Issue #1222⁴⁰⁸² - Merging all the changes
- Issue #1221⁴⁰⁸³ - No error output when using mpirun with hpx
- Issue #1219⁴⁰⁸⁴ - Adding new AGAS cache performance counters
- Issue #1216⁴⁰⁸⁵ - Fixing using futures (clients) as arguments to actions
- Issue #1215⁴⁰⁸⁶ - Error compiling simple component
- Issue #1214⁴⁰⁸⁷ - Stencil docs
- Issue #1213⁴⁰⁸⁸ - Using more than a few dozen MPI processes on SuperMike results in a seg fault before getting to hpx_main
- Issue #1212⁴⁰⁸⁹ - Parallel rotate
- Issue #1211⁴⁰⁹⁰ - Direct actions cause the future's shared_state to be leaked
- Issue #1210⁴⁰⁹¹ - Refactored local::promise to be standard conformant
- Issue #1209⁴⁰⁹² - Improve command line handling
- Issue #1208⁴⁰⁹³ - Adding parallel::reverse and parallel::reverse_copy
- Issue #1207⁴⁰⁹⁴ - Add copy_backward and move_backward
- Issue #1206⁴⁰⁹⁵ - N4071 additional algorithms implemented

⁴⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1232>

⁴⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1231>

⁴⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1230>

⁴⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1229>

⁴⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1228>

⁴⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1226>

⁴⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1225>

⁴⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1224>

⁴⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1223>

⁴⁰⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1222>

⁴⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1221>

⁴⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1219>

⁴⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1216>

⁴⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1215>

⁴⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1214>

⁴⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1213>

⁴⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1212>

⁴⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1211>

⁴⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1210>

⁴⁰⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1209>

⁴⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1208>

⁴⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1207>

⁴⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1206>

- Issue #1204⁴⁰⁹⁶ - Cmake simplification and various other minor changes
- Issue #1203⁴⁰⁹⁷ - Implementing new launch policy for (local) async: `hpx::launch::fork`.
- Issue #1202⁴⁰⁹⁸ - Failed assertion in `connection_cache.hpp`
- Issue #1201⁴⁰⁹⁹ - `pkg-config` doesn't add mpi link directories
- Issue #1200⁴¹⁰⁰ - Error when querying time performance counters
- Issue #1199⁴¹⁰¹ - library path is now configurable (again)
- Issue #1198⁴¹⁰² - Error when querying performance counters
- Issue #1197⁴¹⁰³ - tests fail with intel compiler
- Issue #1196⁴¹⁰⁴ - Silence several warnings
- Issue #1195⁴¹⁰⁵ - Rephrase initializers to work with VC++ 2012
- Issue #1194⁴¹⁰⁶ - Simplify parallel algorithms
- Issue #1193⁴¹⁰⁷ - Adding `parallel::equal`
- Issue #1192⁴¹⁰⁸ - HPX(`out_of_memory`) on including `<hpx/hpx.hpp>`
- Issue #1191⁴¹⁰⁹ - Fixing #1189
- Issue #1190⁴¹¹⁰ - Chrono cleanup
- Issue #1189⁴¹¹¹ - Deadlock .. somewhere? (probably serialization)
- Issue #1188⁴¹¹² - Removed `future::get_status()`
- Issue #1186⁴¹¹³ - Fixed FindOpenCL to find current AMD APP SDK
- Issue #1184⁴¹¹⁴ - Tweaking future unwrapping
- Issue #1183⁴¹¹⁵ - Extended `parallel::reduce`
- Issue #1182⁴¹¹⁶ - `future::unwrap` hangs for `launch::deferred`
- Issue #1181⁴¹¹⁷ - Adding `all_of`, `any_of`, and `none_of` and corresponding documentation
- Issue #1180⁴¹¹⁸ - `hpx::cout` defect

⁴⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1204>

⁴⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1203>

⁴⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1202>

⁴⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1201>

⁴¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1200>

⁴¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1199>

⁴¹⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1198>

⁴¹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1197>

⁴¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1196>

⁴¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1195>

⁴¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1194>

⁴¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1193>

⁴¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1192>

⁴¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1191>

⁴¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1190>

⁴¹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1189>

⁴¹¹² <https://github.com/STELLAR-GROUP/hpx/issues/1188>

⁴¹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1186>

⁴¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1184>

⁴¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1183>

⁴¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1182>

⁴¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1181>

⁴¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1180>

- Issue #1179⁴¹¹⁹ - hpx::async does not work for member function pointers when called on types with self-defined unary operator*
- Issue #1178⁴¹²⁰ - Implemented variadic hpx::util::zip_iterator
- Issue #1177⁴¹²¹ - MPI parcelport defect
- Issue #1176⁴¹²² - HPX_DEFINE_COMPONENT_CONST_ACTION_TPL does not have a 2-argument version
- Issue #1175⁴¹²³ - Create util::zip_iterator working with util::tuple<>
- Issue #1174⁴¹²⁴ - Error Building HPX on linux, root_certificate_authority.cpp
- Issue #1173⁴¹²⁵ - hpx::cout output lost
- Issue #1172⁴¹²⁶ - HPX build error with Clang 3.4.2
- Issue #1171⁴¹²⁷ - CMAKE_INSTALL_PREFIX ignored
- Issue #1170⁴¹²⁸ - Close hpx_benchmarks repository on Github
- Issue #1169⁴¹²⁹ - Buildbot emails have syntax error in url
- Issue #1167⁴¹³⁰ - Merge partial implementation of standards proposal N3960
- Issue #1166⁴¹³¹ - Fixed several compiler warnings
- Issue #1165⁴¹³² - cmake warns: “tests.regressions.actions” does not exist
- Issue #1164⁴¹³³ - Want my own serialization of hpx::future
- Issue #1162⁴¹³⁴ - Segfault in hello_world example
- Issue #1161⁴¹³⁵ - Use HPX_ASSERT to aid the compiler
- Issue #1160⁴¹³⁶ - Do not put -DNDEBUG into hpx_application.pc
- Issue #1159⁴¹³⁷ - Support Clang 3.4.2
- Issue #1158⁴¹³⁸ - Fixed #1157: Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1157⁴¹³⁹ - Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1156⁴¹⁴⁰ - Force inlining fails
- Issue #1155⁴¹⁴¹ - changed header of printout to be compatible with python csv module

⁴¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1179>

⁴¹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1178>

⁴¹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1177>

⁴¹²² <https://github.com/STELLAR-GROUP/hpx/issues/1176>

⁴¹²³ <https://github.com/STELLAR-GROUP/hpx/issues/1175>

⁴¹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1174>

⁴¹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1173>

⁴¹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1172>

⁴¹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1171>

⁴¹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1170>

⁴¹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1169>

⁴¹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1167>

⁴¹³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1166>

⁴¹³² <https://github.com/STELLAR-GROUP/hpx/issues/1165>

⁴¹³³ <https://github.com/STELLAR-GROUP/hpx/issues/1164>

⁴¹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1162>

⁴¹³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1161>

⁴¹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1160>

⁴¹³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1159>

⁴¹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1158>

⁴¹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1157>

⁴¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1156>

⁴¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1155>

- Issue #1154⁴¹⁴² - Fixing iostreams
- Issue #1153⁴¹⁴³ - Standard manipulators (like std::endl) do not work with hpx::ostream
- Issue #1152⁴¹⁴⁴ - Functions revamp
- Issue #1151⁴¹⁴⁵ - Suppressing cmake 3.0 policy warning for CMP0026
- Issue #1150⁴¹⁴⁶ - Client Serialization error
- Issue #1149⁴¹⁴⁷ - Segfault on Stampede
- Issue #1148⁴¹⁴⁸ - Refactoring mini-ghost
- Issue #1147⁴¹⁴⁹ - N3960 copy_if and copy_n implemented and tested
- Issue #1146⁴¹⁵⁰ - Stencil print
- Issue #1145⁴¹⁵¹ - N3960 hpx::parallel::copy implemented and tested
- Issue #1144⁴¹⁵² - OpenMP examples 1d_stencil do not build
- Issue #1143⁴¹⁵³ - 1d_stencil OpenMP examples do not build
- Issue #1142⁴¹⁵⁴ - Cannot build HPX with gcc 4.6 on OS X
- Issue #1140⁴¹⁵⁵ - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
- Issue #1139⁴¹⁵⁶ - hpx/hpx/config/compiler_specific.hpp
- Issue #1138⁴¹⁵⁷ - clean up pkg-config files
- Issue #1137⁴¹⁵⁸ - Improvements to create binary packages
- Issue #1136⁴¹⁵⁹ - HPX_GCC_VERSION not defined on all compilers
- Issue #1135⁴¹⁶⁰ - Avoiding collision between winsock2.h and windows.h
- Issue #1134⁴¹⁶¹ - Making sure, that hpx::finalize can be called from any locality
- Issue #1133⁴¹⁶² - 1d stencil examples
- Issue #1131⁴¹⁶³ - Refactor unique_function implementation
- Issue #1130⁴¹⁶⁴ - Unique function

⁴¹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1154>

⁴¹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1153>

⁴¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1152>

⁴¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1151>

⁴¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1150>

⁴¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1149>

⁴¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1148>

⁴¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1147>

⁴¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1146>

⁴¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1145>

⁴¹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1144>

⁴¹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1143>

⁴¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1142>

⁴¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1140>

⁴¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1139>

⁴¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1138>

⁴¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1137>

⁴¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1136>

⁴¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1135>

⁴¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1134>

⁴¹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1133>

⁴¹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1131>

⁴¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1130>

- Issue #1129⁴¹⁶⁵ - Some fixes to the Build system on OS X
- Issue #1128⁴¹⁶⁶ - Action future args
- Issue #1127⁴¹⁶⁷ - Executor causes segmentation fault
- Issue #1124⁴¹⁶⁸ - Adding new API functions: register_id_with_basename, unregister_id_with_basename, find_ids_from_basename; adding test
- Issue #1123⁴¹⁶⁹ - Reduce nesting of try-catch construct in encode_parcels?
- Issue #1122⁴¹⁷⁰ - Client base fixes
- Issue #1121⁴¹⁷¹ - Update hpxrun.py.in
- Issue #1120⁴¹⁷² - HTTS2 tests compile errors on v110 (VS2012)
- Issue #1119⁴¹⁷³ - Remove references to boost::atomic in accumulator example
- Issue #1118⁴¹⁷⁴ - Only build test thread_pool_executor_1114_test if HPX_LOCAL_SCHEDULER is set
- Issue #1117⁴¹⁷⁵ - local_queue_executor linker error on vc110
- Issue #1116⁴¹⁷⁶ - Disabled performance counter should give runtime errors, not invalid data
- Issue #1115⁴¹⁷⁷ - Compile error with Intel C++ 13.1
- Issue #1114⁴¹⁷⁸ - Default constructed executor is not usable
- Issue #1113⁴¹⁷⁹ - Fast compilation of logging causes ABI incompatibilities between different NDEBUG values
- Issue #1112⁴¹⁸⁰ - Using thread_pool_executors causes segfault
- Issue #1111⁴¹⁸¹ - hpx::threads::get_thread_data always returns zero
- Issue #1110⁴¹⁸² - Remove unnecessary null pointer checks
- Issue #1109⁴¹⁸³ - More tests adjustments
- Issue #1108⁴¹⁸⁴ - Clarify build rules for “libboost_atomic-mt.so”?
- Issue #1107⁴¹⁸⁵ - Remove unnecessary null pointer checks
- Issue #1106⁴¹⁸⁶ - network_storage benchmark improvements, adding legends to plots and tidying layout
- Issue #1105⁴¹⁸⁷ - Add more plot outputs and improve instructions doc

⁴¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1129>

⁴¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1128>

⁴¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1127>

⁴¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1124>

⁴¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1123>

⁴¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1122>

⁴¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1121>

⁴¹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1120>

⁴¹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1119>

⁴¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1118>

⁴¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1117>

⁴¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1116>

⁴¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1115>

⁴¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1114>

⁴¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1113>

⁴¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1112>

⁴¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1111>

⁴¹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1110>

⁴¹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1109>

⁴¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1108>

⁴¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1107>

⁴¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1106>

⁴¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1105>

- Issue #1104⁴¹⁸⁸ - Complete quoting for parameters of some CMake commands
- Issue #1103⁴¹⁸⁹ - Work on test/scripts
- Issue #1102⁴¹⁹⁰ - Changed minimum requirement of window install to 2012
- Issue #1101⁴¹⁹¹ - Changed minimum requirement of window install to 2012
- Issue #1100⁴¹⁹² - Changed readme to no longer specify using MSVC 2010 compiler
- Issue #1099⁴¹⁹³ - Error returning futures from component actions
- Issue #1098⁴¹⁹⁴ - Improve storage test
- Issue #1097⁴¹⁹⁵ - data_actions quickstart example calls missing function decorate_action of data_get_action
- Issue #1096⁴¹⁹⁶ - MPI parcelport broken with new zero copy optimization
- Issue #1095⁴¹⁹⁷ - Warning C4005: _WIN32_WINNT: Macro redefinition
- Issue #1094⁴¹⁹⁸ - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
- Issue #1093⁴¹⁹⁹ - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS
- Issue #1092⁴²⁰⁰ - Rename unique_future<> back to future<>
- Issue #1091⁴²⁰¹ - Inconsistent error message
- Issue #1090⁴²⁰² - On windows 8.1 the examples crashed if using more than one os thread
- Issue #1089⁴²⁰³ - Components should be allowed to have their own executor
- Issue #1088⁴²⁰⁴ - Add possibility to select a network interface for the ibverbs parcelport
- Issue #1087⁴²⁰⁵ - ibverbs and ipc parcelport uses zero copy optimization
- Issue #1083⁴²⁰⁶ - Make shell examples copyable in docs
- Issue #1082⁴²⁰⁷ - Implement proper termination detection during shutdown
- Issue #1081⁴²⁰⁸ - Implement thread_specific_ptr for hpx::threads
- Issue #1072⁴²⁰⁹ - make install not working properly
- Issue #1070⁴²¹⁰ - Complete quoting for parameters of some CMake commands

⁴¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1104>

⁴¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1103>

⁴¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1102>

⁴¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1101>

⁴¹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1100>

⁴¹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1099>

⁴¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1098>

⁴¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1097>

⁴¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1096>

⁴¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1095>

⁴¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1094>

⁴¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1093>

⁴²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1092>

⁴²⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1091>

⁴²⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1090>

⁴²⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1089>

⁴²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1088>

⁴²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1087>

⁴²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1083>

⁴²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1082>

⁴²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1081>

⁴²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1072>

⁴²¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1070>

- Issue #1059⁴²¹¹ - Fix more unused variable warnings
- Issue #1051⁴²¹² - Implement when_each
- Issue #973⁴²¹³ - Would like option to report hwloc bindings
- Issue #970⁴²¹⁴ - Bad flags for Fortran compiler
- Issue #941⁴²¹⁵ - Create a proper user level context switching class for BG/Q
- Issue #935⁴²¹⁶ - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #934⁴²¹⁷ - Want to build HPX without dynamic libraries
- Issue #927⁴²¹⁸ - Make hpx/lcos/reduce.hpp accept futures of id_type
- Issue #926⁴²¹⁹ - All unit tests that are run with more than one thread with CTest/hpx_run_test should configure hpx.os_threads
- Issue #925⁴²²⁰ - regression_dataflow_791 needs to be brought in line with HPX standards
- Issue #899⁴²²¹ - Fix race conditions in regression tests
- Issue #879⁴²²² - Hung test leads to cascading test failure; make tests should support the MPI parcelport
- Issue #865⁴²²³ - future<T> and friends shall work for movable only Ts
- Issue #847⁴²²⁴ - Dynamic libraries are not installed on OS X
- Issue #816⁴²²⁵ - First Program tutorial pull request
- Issue #799⁴²²⁶ - Wrap lexical_cast to avoid exceptions
- Issue #720⁴²²⁷ - broken configuration when using ccmake on Ubuntu
- Issue #622⁴²²⁸ - --hpx:hpx and --hpx:debug-hpx-log is nonsensical
- Issue #525⁴²²⁹ - Extend barrier LCO test to run in distributed
- Issue #515⁴²³⁰ - Multi-destination version of hpx::apply is broken
- Issue #509⁴²³¹ - Push Boost.Atomic changes upstream
- Issue #503⁴²³² - Running HPX applications on Windows should not require setting %PATH%
- Issue #461⁴²³³ - Add a compilation sanity test

⁴²¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1059>

⁴²¹² <https://github.com/STELLAR-GROUP/hpx/issues/1051>

⁴²¹³ <https://github.com/STELLAR-GROUP/hpx/issues/973>

⁴²¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/970>

⁴²¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/941>

⁴²¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/935>

⁴²¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/934>

⁴²¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/927>

⁴²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/926>

⁴²²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/925>

4221 <https://github.com/STELLAR-GROUP/hpx/issues/899>4222 <https://github.com/STELLAR-GROUP/hpx/issues/879>4223 <https://github.com/STELLAR-GROUP/hpx/issues/865>4224 <https://github.com/STELLAR-GROUP/hpx/issues/847>4225 <https://github.com/STELLAR-GROUP/hpx/issues/816>4226 <https://github.com/STELLAR-GROUP/hpx/issues/799>4227 <https://github.com/STELLAR-GROUP/hpx/issues/720>4228 <https://github.com/STELLAR-GROUP/hpx/issues/622>4229 <https://github.com/STELLAR-GROUP/hpx/issues/525>4230 <https://github.com/STELLAR-GROUP/hpx/issues/515>4231 <https://github.com/STELLAR-GROUP/hpx/issues/509>4232 <https://github.com/STELLAR-GROUP/hpx/issues/503>4233 <https://github.com/STELLAR-GROUP/hpx/issues/461>

- Issue #456⁴²³⁴ - hpx_run_tests.py should log output from tests that timeout
- Issue #454⁴²³⁵ - Investigate threadmanager performance
- Issue #345⁴²³⁶ - Add more versatile environmental/cmake variable support to hpx_find_* CMake macros
- Issue #209⁴²³⁷ - Support multiple configurations in generated build files
- Issue #190⁴²³⁸ - hpx::cout should be a std::ostream
- Issue #189⁴²³⁹ - iostreams component should use startup/shutdown functions
- Issue #183⁴²⁴⁰ - Use Boost.ICL for correctness in AGAS
- Issue #44⁴²⁴¹ - Implement real futures

2.10.17 HPX V0.9.8 (Mar 24, 2014)

We have had over 800 commits since the last release and we have closed over 65 tickets (bugs, feature requests, etc.).

With the changes below, *HPX* is once again leading the charge of a whole new era of computation. By intrinsically breaking down and synchronizing the work to be done, *HPX* insures that application developers will no longer have to fret about where a segment of code executes. That allows coders to focus their time and energy to understanding the data dependencies of their algorithms and thereby the core obstacles to an efficient code. Here are some of the advantages of using *HPX*:

- *HPX* is solidly rooted in a sophisticated theoretical execution model – ParalleX
- *HPX* exposes an API fully conforming to the C++11 and the draft C++14 standards, extended and applied to distributed computing. Everything programmers know about the concurrency primitives of the standard C++ library is still valid in the context of *HPX*.
- It provides a competitive, high performance implementation of modern, future-proof ideas which gives an smooth migration path from today's mainstream techniques
- There is no need for the programmer to worry about lower level parallelization paradigms like threads or message passing; no need to understand pthreads, MPI, OpenMP, or Windows threads, etc.
- There is no need to think about different types of parallelism such as tasks, pipelines, or fork-join, task or data parallelism.
- The same source of your program compiles and runs on Linux, BlueGene/Q, Mac OS X, Windows, and Android.
- The same code runs on shared memory multi-core systems and supercomputers, on handheld devices and Intel® Xeon Phi™ accelerators, or a heterogeneous mix of those.

⁴²³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/456>

⁴²³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/454>

⁴²³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/345>

⁴²³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/209>

4238 <https://github.com/STELLAR-GROUP/hpx/issues/190>4239 <https://github.com/STELLAR-GROUP/hpx/issues/189>4240 <https://github.com/STELLAR-GROUP/hpx/issues/183>4241 <https://github.com/STELLAR-GROUP/hpx/issues/44>

General changes

- A major API breaking change for this release was introduced by implementing `hpx::future` and `hpx::shared_future` fully in conformance with the C++11 Standard⁴²⁴². While `hpx::shared_future` is new and will not create any compatibility problems, we revised the interface and implementation of the existing `hpx::future`. For more details please see the mailing list archive⁴²⁴³. 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 Standard⁴²⁴⁴.
- 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 at 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 specification⁴²⁴⁵.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1086⁴²⁴⁶ - Expose internal boost::shared_array to allow user management of array lifetime
- Issue #1083⁴²⁴⁷ - Make shell examples copyable in docs
- Issue #1080⁴²⁴⁸ - /threads{locality#*/total}/count/cumulative broken

⁴²⁴² <http://www.open-std.org/jtc1/sc22/wg21>

⁴²⁴³ <http://mail.cct.lsu.edu/pipermail/hpx-users/2014-January/000141.html>

⁴²⁴⁴ <http://www.open-std.org/jtc1/sc22/wg21>

⁴²⁴⁵ <https://github.com/STELLAR-GROUP/hpxpi/blob/master/spec.pdf?raw=true>

⁴²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1086>

⁴²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1083>

⁴²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1080>

- Issue #1079⁴²⁴⁹ - Build problems on OS X
- Issue #1078⁴²⁵⁰ - Improve robustness of component loading
- Issue #1077⁴²⁵¹ - Fix a missing enum definition for ‘take’ mode
- Issue #1076⁴²⁵² - Merge Jb master
- Issue #1075⁴²⁵³ - Unknown CMake command “add_hpx_pseudo_target”
- Issue #1074⁴²⁵⁴ - Implement apply_continue_callback and apply_colocated_callback
- Issue #1073⁴²⁵⁵ - The new apply_colocated and async_colocated functions lead to automatic registered functions
- Issue #1071⁴²⁵⁶ - Remove deferred_packaged_task
- Issue #1069⁴²⁵⁷ - serialize_buffer with allocator fails at destruction
- Issue #1068⁴²⁵⁸ - Coroutine include and forward declarations missing
- Issue #1067⁴²⁵⁹ - Add allocator support to util::serialize_buffer
- Issue #1066⁴²⁶⁰ - Allow for MPI_Init being called before HPX launches
- Issue #1065⁴²⁶¹ - AGAS cache isn’t used/populated on worker localities
- Issue #1064⁴²⁶² - Reorder includes to ensure ws2 includes early
- Issue #1063⁴²⁶³ - Add hpx::runtime::suspend and hpx::runtime::resume
- Issue #1062⁴²⁶⁴ - Fix async_continue to properly handle return types
- Issue #1061⁴²⁶⁵ - Implement async_colocated and apply_colocated
- Issue #1060⁴²⁶⁶ - Implement minimal component migration
- Issue #1058⁴²⁶⁷ - Remove HPX_UTIL_TUPLE from code base
- Issue #1057⁴²⁶⁸ - Add performance counters for threading subsystem
- Issue #1055⁴²⁶⁹ - Thread allocation uses two memory pools
- Issue #1053⁴²⁷⁰ - Work stealing flawed
- Issue #1052⁴²⁷¹ - Fix a number of warnings

⁴²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1079>

⁴²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1078>

⁴²⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1077>

⁴²⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1076>

⁴²⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1075>

⁴²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1074>

⁴²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1073>

⁴²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1071>

⁴²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1069>

⁴²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1068>

⁴²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1067>

⁴²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1066>

⁴²⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1065>

⁴²⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1064>

⁴²⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1063>

⁴²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1062>

⁴²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1061>

⁴²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1060>

⁴²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1058>

⁴²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1057>

⁴²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1055>

⁴²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1053>

⁴²⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1052>

- Issue #1049⁴²⁷² - Fixes for TLS on OSX and more reliable test running
- Issue #1048⁴²⁷³ - Fixing after 588 hang
- Issue #1047⁴²⁷⁴ - Use port ‘0’ for networking when using one locality
- Issue #1046⁴²⁷⁵ - composable_guard test is broken when having more than one thread
- Issue #1045⁴²⁷⁶ - Security missing headers
- Issue #1044⁴²⁷⁷ - Native TLS on FreeBSD via __thread
- Issue #1043⁴²⁷⁸ - async et.al. compute the wrong result type
- Issue #1042⁴²⁷⁹ - async et.al. implicitly unwrap reference_wrappers
- Issue #1041⁴²⁸⁰ - Remove redundant costly Kleene stars from regex searches
- Issue #1040⁴²⁸¹ - CMake script regex match patterns has unnecessary kleenes
- Issue #1039⁴²⁸² - Remove use of Boost.Move and replace with std::move and real rvalue refs
- Issue #1038⁴²⁸³ - Bump minimal required Boost to 1.49.0
- Issue #1037⁴²⁸⁴ - Implicit unwrapping of futures in async broken
- Issue #1036⁴²⁸⁵ - Scheduler hangs when user code attempts to “block” OS-threads
- Issue #1035⁴²⁸⁶ - Idle-rate counter always reports 100% idle rate
- Issue #1034⁴²⁸⁷ - Symbolic name registration causes application hangs
- Issue #1033⁴²⁸⁸ - Application options read in from an options file generate an error message
- Issue #1032⁴²⁸⁹ - hpx::id_type local reference counting is wrong
- Issue #1031⁴²⁹⁰ - Negative entry in reference count table
- Issue #1030⁴²⁹¹ - Implement condition_variable
- Issue #1029⁴²⁹² - Deadlock in thread scheduling subsystem
- Issue #1028⁴²⁹³ - HPX-thread cumulative count performance counters report incorrect value
- Issue #1027⁴²⁹⁴ - Expose hpx::thread_interrupted error code as a separate exception type

⁴²⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1049>

⁴²⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1048>

⁴²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1047>

⁴²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1046>

⁴²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1045>

⁴²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1044>

⁴²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1043>

⁴²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1042>

⁴²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1041>

⁴²⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1040>

⁴²⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1039>

⁴²⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1038>

⁴²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1037>

⁴²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1036>

⁴²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1035>

⁴²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1034>

⁴²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1033>

⁴²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1032>

⁴²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1031>

⁴²⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1030>

⁴²⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1029>

⁴²⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1028>

⁴²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1027>

- Issue #1026⁴²⁹⁵ - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
- Issue #1025⁴²⁹⁶ - `future::wait_for/wait_until` do not remove callback
- Issue #1024⁴²⁹⁷ - Remove dependence to boost assert and create hpx assert
- Issue #1023⁴²⁹⁸ - Segfaults with tcmalloc
- Issue #1022⁴²⁹⁹ - prerequisites link in readme is broken
- Issue #1020⁴³⁰⁰ - HPX Deadlock on external synchronization
- Issue #1019⁴³⁰¹ - Convert using `BOOST_ASSERT` to `HPX_ASSERT`
- Issue #1018⁴³⁰² - compiling bug with gcc 4.8.1
- Issue #1017⁴³⁰³ - Possible crash in io_pool executor
- Issue #1016⁴³⁰⁴ - Crash at startup
- Issue #1014⁴³⁰⁵ - Implement Increment/Decrement Merging
- Issue #1013⁴³⁰⁶ - Add more logging channels to enable greater control over logging granularity
- Issue #1012⁴³⁰⁷ - `--hpx:debug-hpx-log` and `--hpx:debug-agas-log` lead to non-thread safe writes
- Issue #1011⁴³⁰⁸ - After installation, running applications from the build/staging directory no longer works
- Issue #1010⁴³⁰⁹ - Mergeable decrement requests are not being merged
- Issue #1009⁴³¹⁰ - `--hpx:list-symbolic-names` crashes
- Issue #1007⁴³¹¹ - Components are not properly destroyed
- Issue #1006⁴³¹² - Segfault/hang in `set_data`
- Issue #1003⁴³¹³ - Performance counter naming issue
- Issue #982⁴³¹⁴ - Race condition during startup
- Issue #912⁴³¹⁵ - OS X: component type not found in map
- Issue #663⁴³¹⁶ - Create a buildbot slave based on Clang 3.2/OSX
- Issue #636⁴³¹⁷ - Expose `this_locality::apply<act>(p1, p2);` for local execution

⁴²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1026>

⁴²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1025>

⁴²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1024>

⁴²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1023>

⁴²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1022>

⁴³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1020>

⁴³⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1019>

⁴³⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1018>

⁴³⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1017>

⁴³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1016>

⁴³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1014>

⁴³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1013>

⁴³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1012>

⁴³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1011>

⁴³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1010>

⁴³¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1009>

⁴³¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1007>

⁴³¹² <https://github.com/STELLAR-GROUP/hpx/issues/1006>

⁴³¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1003>

⁴³¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/982>

⁴³¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/912>

⁴³¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/663>

⁴³¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/636>

- Issue #197⁴³¹⁸ - Add --console=address option for PBS runs
- Issue #175⁴³¹⁹ - Asynchronous AGAS API

2.10.18 HPX V0.9.7 (Nov 13, 2013)

We have had over 1000 commits since the last release and we have closed over 180 tickets (bugs, feature requests, etc.).

General changes

- Ported HPX to BlueGene/Q
- Improved HPX support for Xeon/Phi accelerators
- Reimplemented `hpx::bind`, `hpx::tuple`, and `hpx::function` for better performance and better compliance with the C++11 Standard. Added `hpx::mem_fn`.
- Reworked `hpx::when_all` and `hpx::when_any` for better compliance with the ongoing C++ standardization effort, added heterogeneous version for those functions. Added `hpx::when_any_swapped`.
- Added `hpx::copy` as a precursor for a migrate functionality
- Added `hpx::get_ptr` allowing to directly access the memory underlying a given component
- Added the `hpx::lcos::broadcast`, `hpx::lcos::reduce`, and `hpx::lcos::fold` collective operations
- Added `hpx::get_locality_name` allowing to retrieve the name of any of the localities for the application.
- Added support for more flexible thread affinity control from the HPX command line, such as new modes for `--hpx:bind` (balanced, scattered, compact), improved default settings when running multiple localities on the same node.
- Added experimental executors for simpler thread pooling and scheduling. This API may change in the future as it will stay aligned with the ongoing C++ standardization efforts.
- Massively improved the performance of the HPX serialization code. Added partial support for zero copy serialization of array and bitwise-copyable types.
- General performance improvements of the code related to threads and futures.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1005⁴³²⁰ - Allow one to disable array optimizations and zero copy optimizations for each parcelport
- Issue #1004⁴³²¹ - Generate new HPX logo image for the docs
- Issue #1002⁴³²² - If MPI parcelport is not available, running HPX under mpirun should fail
- Issue #1001⁴³²³ - Zero copy serialization raises assert

⁴³¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/197>

⁴³¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/175>

⁴³²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1005>

⁴³²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1004>

⁴³²² <https://github.com/STELLAR-GROUP/hpx/issues/1002>

⁴³²³ <https://github.com/STELLAR-GROUP/hpx/issues/1001>

- Issue #1000⁴³²⁴ - Can't connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
- Issue #999⁴³²⁵ - Optimize hpx::when_n
- Issue #998⁴³²⁶ - Fixed const-correctness
- Issue #997⁴³²⁷ - Making serialize_buffer::data() type save
- Issue #996⁴³²⁸ - Memory leak in hpx::lcos::promise
- Issue #995⁴³²⁹ - Race while registering pre-shutdown functions
- Issue #994⁴³³⁰ - thread_rescheduling regression test does not compile
- Issue #992⁴³³¹ - Correct comments and messages
- Issue #991⁴³³² - setcap cap_sys_rawio=ep for power profiling causes an HPX application to abort
- Issue #989⁴³³³ - Jacobi hangs during execution
- Issue #988⁴³³⁴ - multiple_init test is failing
- Issue #986⁴³³⁵ - Can't call a function called "init" from "main" when using <hpx/hpx_main.hpp>
- Issue #984⁴³³⁶ - Reference counting tests are failing
- Issue #983⁴³³⁷ - thread_suspension_executor test fails
- Issue #980⁴³³⁸ - Terminating HPX threads don't leave stack in virgin state
- Issue #979⁴³³⁹ - Static scheduler not in documents
- Issue #978⁴³⁴⁰ - Preprocessing limits are broken
- Issue #977⁴³⁴¹ - Make tests.regressions.lcos.future_hang_on_get shorter
- Issue #976⁴³⁴² - Wrong library order in pkgconfig
- Issue #975⁴³⁴³ - Please reopen #963
- Issue #974⁴³⁴⁴ - Option pu-offset ignored in fixing_588 branch
- Issue #972⁴³⁴⁵ - Cannot use MKL with HPX
- Issue #969⁴³⁴⁶ - Non-existent INI files requested on the command line via --hpx:config do not cause warn-

⁴³²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1000>

⁴³²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/999>

⁴³²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/998>

⁴³²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/997>

⁴³²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/996>

⁴³²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/995>

⁴³³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/994>

⁴³³¹ <https://github.com/STELLAR-GROUP/hpx/issues/992>

⁴³³² <https://github.com/STELLAR-GROUP/hpx/issues/991>

⁴³³³ <https://github.com/STELLAR-GROUP/hpx/issues/989>

⁴³³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/988>

⁴³³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/986>

⁴³³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/984>

⁴³³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/983>

⁴³³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/980>

⁴³³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/979>

⁴³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/978>

⁴³⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/977>

⁴³⁴² <https://github.com/STELLAR-GROUP/hpx/issues/976>

⁴³⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/975>

⁴³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/974>

⁴³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/972>

⁴³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/969>

ings or errors.

- Issue #968⁴³⁴⁷ - Cannot build examples in fixing_588 branch
- Issue #967⁴³⁴⁸ - Command line description of --hpx:queuing seems wrong
- Issue #966⁴³⁴⁹ - --hpx:print-bind physical core numbers are wrong
- Issue #965⁴³⁵⁰ - Deadlock when building in Release mode
- Issue #963⁴³⁵¹ - Not all worker threads are working
- Issue #962⁴³⁵² - Problem with SLURM integration
- Issue #961⁴³⁵³ - --hpx:print-bind outputs incorrect information
- Issue #960⁴³⁵⁴ - Fix cut and paste error in documentation of get_thread_priority
- Issue #959⁴³⁵⁵ - Change link to boost.atomic in documentation to point to boost.org
- Issue #958⁴³⁵⁶ - Undefined reference to intrusive_ptr_release
- Issue #957⁴³⁵⁷ - Make tuple standard compliant
- Issue #956⁴³⁵⁸ - Segfault with a3382fb
- Issue #955⁴³⁵⁹ - --hpx:nodes and --hpx:nodefiles do not work with foreign nodes
- Issue #954⁴³⁶⁰ - Make order of arguments for hpx::async and hpx::broadcast consistent
- Issue #953⁴³⁶¹ - Cannot use MKL with HPX
- Issue #952⁴³⁶² - register_[pre_] shutdown_function never throw
- Issue #951⁴³⁶³ - Assert when number of threads is greater than hardware concurrency
- Issue #948⁴³⁶⁴ - HPX_HAVE_GENERIC_CONTEXT_COROUTINES conflicts with HPX_HAVE_FIBER_BASED_COROUTINES
- Issue #947⁴³⁶⁵ - Need MPI_THREAD_MULTIPLE for backward compatibility
- Issue #946⁴³⁶⁶ - HPX does not call MPI_Finalize
- Issue #945⁴³⁶⁷ - Segfault with hpx::lcos::broadcast
- Issue #944⁴³⁶⁸ - OS X: assertion pu_offset_ < hardware_concurrency failed

⁴³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/968>

⁴³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/967>

⁴³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/966>

⁴³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/965>

⁴³⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/963>

⁴³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/962>

⁴³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/961>

⁴³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/960>

⁴³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/959>

⁴³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/958>

⁴³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/957>

⁴³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/956>

⁴³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/955>

⁴³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/954>

⁴³⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/953>

⁴³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/952>

⁴³⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/951>

⁴³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/948>

⁴³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/947>

⁴³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/946>

⁴³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/945>

⁴³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/944>

- Issue #943⁴³⁶⁹ - #include <hpx/hpx_main.hpp> does not work
- Issue #942⁴³⁷⁰ - Make the BG/Q work with -O3
- Issue #940⁴³⁷¹ - Use separator when concatenating locality name
- Issue #939⁴³⁷² - Refactor MPI parcelport to use MPI_Wait instead of multiple MPI_Test calls
- Issue #938⁴³⁷³ - Want to officially access client_base::gid_
- Issue #937⁴³⁷⁴ - client_base::gid_ should be private``
- Issue #936⁴³⁷⁵ - Want doxygen-like source code index
- Issue #935⁴³⁷⁶ - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #933⁴³⁷⁷ - Cannot build HPX with Boost 1.54.0
- Issue #932⁴³⁷⁸ - Components are destructed too early
- Issue #931⁴³⁷⁹ - Make HPX work on BG/Q
- Issue #930⁴³⁸⁰ - make git-docs is broken
- Issue #929⁴³⁸¹ - Generating index in docs broken
- Issue #928⁴³⁸² - Optimize hpx::util::static_ for C++11 compilers supporting magic statics
- Issue #924⁴³⁸³ - Make kill_process_tree (in process.py) more robust on Mac OSX
- Issue #923⁴³⁸⁴ - Correct BLAS and RNPL cmake tests
- Issue #922⁴³⁸⁵ - Cannot link against BLAS
- Issue #921⁴³⁸⁶ - Implement hpx::mem_fn
- Issue #920⁴³⁸⁷ - Output locality with --hpx:print-bind
- Issue #919⁴³⁸⁸ - Correct grammar; simplify boolean expressions
- Issue #918⁴³⁸⁹ - Link to hello_world.cpp is broken
- Issue #917⁴³⁹⁰ - adapt cmake file to new boostbook version
- Issue #916⁴³⁹¹ - fix problem building documentation with xsltproc >= 1.1.27

⁴³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/943>

⁴³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/942>

⁴³⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/940>

⁴³⁷² <https://github.com/STELLAR-GROUP/hpx/issues/939>

⁴³⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/938>

⁴³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/937>

⁴³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/936>

⁴³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/935>

⁴³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/933>

⁴³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/932>

⁴³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/931>

⁴³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/930>

⁴³⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/929>

⁴³⁸² <https://github.com/STELLAR-GROUP/hpx/issues/928>

⁴³⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/924>

⁴³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/923>

⁴³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/922>

⁴³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/921>

⁴³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/920>

⁴³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/919>

⁴³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/918>

⁴³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/917>

⁴³⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/916>

- Issue #915⁴³⁹² - Add another TBBMalloc library search path
- Issue #914⁴³⁹³ - Build problem with Intel compiler on Stampede (TACC)
- Issue #913⁴³⁹⁴ - fix error messages in fibonacci examples
- Issue #911⁴³⁹⁵ - Update OS X build instructions
- Issue #910⁴³⁹⁶ - Want like to specify MPI_ROOT instead of compiler wrapper script
- Issue #909⁴³⁹⁷ - Warning about void* arithmetic
- Issue #908⁴³⁹⁸ - Buildbot for MIC is broken
- Issue #906⁴³⁹⁹ - Can't use --hpx:bind=balanced with multiple MPI processes
- Issue #905⁴⁴⁰⁰ - --hpx:bind documentation should describe full grammar
- Issue #904⁴⁴⁰¹ - Add hpx::lcos::fold and hpx::lcos::inverse_fold collective operation
- Issue #903⁴⁴⁰² - Add hpx::when_any_swapped()
- Issue #902⁴⁴⁰³ - Add hpx::lcos::reduce collective operation
- Issue #901⁴⁴⁰⁴ - Web documentation is not searchable
- Issue #900⁴⁴⁰⁵ - Web documentation for trunk has no index
- Issue #898⁴⁴⁰⁶ - Some tests fail with GCC 4.8.1 and MPI parcel port
- Issue #897⁴⁴⁰⁷ - HWLOC causes failures on Mac
- Issue #896⁴⁴⁰⁸ - pu-offset leads to startup error
- Issue #895⁴⁴⁰⁹ - hpx::get_locality_name not defined
- Issue #894⁴⁴¹⁰ - Race condition at shutdown
- Issue #893⁴⁴¹¹ - --hpx:print-bind switches std::cout to hexadecimal mode
- Issue #892⁴⁴¹² - hwloc_topology_load can be expensive – don't call multiple times
- Issue #891⁴⁴¹³ - The documentation for get_locality_name is wrong
- Issue #890⁴⁴¹⁴ - --hpx:print-bind should not exit

⁴³⁹² <https://github.com/STELLAR-GROUP/hpx/issues/915>

⁴³⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/914>

⁴³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/913>

⁴³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/911>

⁴³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/910>

⁴³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/909>

4398 <https://github.com/STELLAR-GROUP/hpx/issues/908>4399 <https://github.com/STELLAR-GROUP/hpx/issues/906>4400 <https://github.com/STELLAR-GROUP/hpx/issues/905>4401 <https://github.com/STELLAR-GROUP/hpx/issues/904>4402 <https://github.com/STELLAR-GROUP/hpx/issues/903>4403 <https://github.com/STELLAR-GROUP/hpx/issues/902>4404 <https://github.com/STELLAR-GROUP/hpx/issues/901>4405 <https://github.com/STELLAR-GROUP/hpx/issues/900>4406 <https://github.com/STELLAR-GROUP/hpx/issues/898>4407 <https://github.com/STELLAR-GROUP/hpx/issues/897>4408 <https://github.com/STELLAR-GROUP/hpx/issues/896>4409 <https://github.com/STELLAR-GROUP/hpx/issues/895>4410 <https://github.com/STELLAR-GROUP/hpx/issues/894>4411 <https://github.com/STELLAR-GROUP/hpx/issues/893>4412 <https://github.com/STELLAR-GROUP/hpx/issues/892>4413 <https://github.com/STELLAR-GROUP/hpx/issues/891>4414 <https://github.com/STELLAR-GROUP/hpx/issues/890>

- Issue #889⁴⁴¹⁵ --hpx:debug-hpx-log=FILE does not work
- Issue #888⁴⁴¹⁶ - MPI parcelport does not exit cleanly for -hpx:print-bind
- Issue #887⁴⁴¹⁷ - Choose thread affinities more cleverly
- Issue #886⁴⁴¹⁸ - Logging documentation is confusing
- Issue #885⁴⁴¹⁹ - Two threads are slower than one
- Issue #884⁴⁴²⁰ - is_callable failing with member pointers in C++11
- Issue #883⁴⁴²¹ - Need help with is_callable_test
- Issue #882⁴⁴²² - tests.regressions.lcos.future_hang_on_get does not terminate
- Issue #881⁴⁴²³ - tests/regressions/block_matrix/matrix.hh won't compile with GCC 4.8.1
- Issue #880⁴⁴²⁴ - HPX does not work on OS X
- Issue #878⁴⁴²⁵ - future::unwrap triggers assertion
- Issue #877⁴⁴²⁶ - “make tests” has build errors on Ubuntu 12.10
- Issue #876⁴⁴²⁷ - tcmalloc is used by default, even if it is not present
- Issue #875⁴⁴²⁸ - global_fixture is defined in a header file
- Issue #874⁴⁴²⁹ - Some tests take very long
- Issue #873⁴⁴³⁰ - Add block-matrix code as regression test
- Issue #872⁴⁴³¹ - HPX documentation does not say how to run tests with detailed output
- Issue #871⁴⁴³² - All tests fail with “make test”
- Issue #870⁴⁴³³ - Please explicitly disable serialization in classes that don't support it
- Issue #868⁴⁴³⁴ - boost_any test failing
- Issue #867⁴⁴³⁵ - Reduce the number of copies of hpx::function arguments
- Issue #863⁴⁴³⁶ - Futures should not require a default constructor
- Issue #862⁴⁴³⁷ - value_or_error shall not default construct its result

⁴⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/889>

⁴⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/888>

⁴⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/887>

⁴⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/886>

⁴⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/885>

⁴⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/884>

⁴⁴²¹ <https://github.com/STELLAR-GROUP/hpx/issues/883>

⁴⁴²² <https://github.com/STELLAR-GROUP/hpx/issues/882>

⁴⁴²³ <https://github.com/STELLAR-GROUP/hpx/issues/881>

⁴⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/880>

⁴⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/878>

⁴⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/877>

⁴⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/876>

⁴⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/875>

⁴⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/874>

⁴⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/873>

⁴⁴³¹ <https://github.com/STELLAR-GROUP/hpx/issues/872>

⁴⁴³² <https://github.com/STELLAR-GROUP/hpx/issues/871>

⁴⁴³³ <https://github.com/STELLAR-GROUP/hpx/issues/870>

⁴⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/868>

⁴⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/867>

⁴⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/863>

⁴⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/862>

- Issue #861⁴⁴³⁸ - HPX_UNUSED macro
- Issue #860⁴⁴³⁹ - Add functionality to copy construct a component
- Issue #859⁴⁴⁴⁰ - hpx::endl should flush
- Issue #858⁴⁴⁴¹ - Create hpx::get_ptr<> allowing to access component implementation
- Issue #855⁴⁴⁴² - Implement hpx::INVOKER
- Issue #854⁴⁴⁴³ - hpx/hpx.hpp does not include hpx/include/iostreams.hpp
- Issue #853⁴⁴⁴⁴ - Feature request: null future
- Issue #852⁴⁴⁴⁵ - Feature request: Locality names
- Issue #851⁴⁴⁴⁶ - hpx::cout output does not appear on screen
- Issue #849⁴⁴⁴⁷ - All tests fail on OS X after installing
- Issue #848⁴⁴⁴⁸ - Update OS X build instructions
- Issue #846⁴⁴⁴⁹ - Update hpx_external_example
- Issue #845⁴⁴⁵⁰ - Issues with having both debug and release modules in the same directory
- Issue #844⁴⁴⁵¹ - Create configuration header
- Issue #843⁴⁴⁵² - Tests should use CTest
- Issue #842⁴⁴⁵³ - Remove buffer_pool from MPI parcelport
- Issue #841⁴⁴⁵⁴ - Add possibility to broadcast an index with hpx::lcos::broadcast
- Issue #838⁴⁴⁵⁵ - Simplify util::tuple
- Issue #837⁴⁴⁵⁶ - Adopt boost::tuple tests for util::tuple
- Issue #836⁴⁴⁵⁷ - Adopt boost::function tests for util::function
- Issue #835⁴⁴⁵⁸ - Tuple interface missing pieces
- Issue #833⁴⁴⁵⁹ - Partially preprocessing files not working
- Issue #832⁴⁴⁶⁰ - Native papi counters do not work with wild cards

⁴⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/861>

⁴⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/860>

⁴⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/859>

⁴⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/858>

⁴⁴⁴² <https://github.com/STELLAR-GROUP/hpx/issues/855>

⁴⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/854>

⁴⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/853>

⁴⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/852>

⁴⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/851>

⁴⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/849>

⁴⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/848>

⁴⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/846>

⁴⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/845>

⁴⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/844>

4452 <https://github.com/STELLAR-GROUP/hpx/issues/843>4453 <https://github.com/STELLAR-GROUP/hpx/issues/842>4454 <https://github.com/STELLAR-GROUP/hpx/issues/841>4455 <https://github.com/STELLAR-GROUP/hpx/issues/838>4456 <https://github.com/STELLAR-GROUP/hpx/issues/837>4457 <https://github.com/STELLAR-GROUP/hpx/issues/836>4458 <https://github.com/STELLAR-GROUP/hpx/issues/835>4459 <https://github.com/STELLAR-GROUP/hpx/issues/833>4460 <https://github.com/STELLAR-GROUP/hpx/issues/832>

- Issue #831⁴⁴⁶¹ - Arithmetics counter fails if only one parameter is given
- Issue #830⁴⁴⁶² - Convert hpx::util::function to use new scheme for serializing its base pointer
- Issue #829⁴⁴⁶³ - Consistently use `decay<T>` instead of `remove_const< remove_reference<T>>`
- Issue #828⁴⁴⁶⁴ - Update future implementation to N3721 and N3722
- Issue #827⁴⁴⁶⁵ - Enable MPI parcelport for bootstrapping whenever application was started using `mpirun`
- Issue #826⁴⁴⁶⁶ - Support command line option `--hpx:print-bind` even if `--hpx::bind` was not used
- Issue #825⁴⁴⁶⁷ - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- Issue #824⁴⁴⁶⁸ - Enable lambda functions to be used with `hpx::async/hpx::apply`
- Issue #823⁴⁴⁶⁹ - Using a hashing filter
- Issue #822⁴⁴⁷⁰ - Silence unused variable warning
- Issue #821⁴⁴⁷¹ - Detect if a function object is callable with given arguments
- Issue #820⁴⁴⁷² - Allow wildcards to be used for performance counter names
- Issue #819⁴⁴⁷³ - Make the AGAS symbolic name registry distributed
- Issue #818⁴⁴⁷⁴ - Add `future::then()` overload taking an executor
- Issue #817⁴⁴⁷⁵ - Fixed typo
- Issue #815⁴⁴⁷⁶ - Create an lco that is performing an efficient broadcast of actions
- Issue #814⁴⁴⁷⁷ - Papi counters cannot specify `thread#*` to get the counts for all threads
- Issue #813⁴⁴⁷⁸ - Scoped unlock
- Issue #811⁴⁴⁷⁹ - `simple_central_tuplespace_client` run error
- Issue #810⁴⁴⁸⁰ - `ostream` error when `<< any objects`
- Issue #809⁴⁴⁸¹ - Optimize parcel serialization
- Issue #808⁴⁴⁸² - HPX applications throw exception when executed from the build directory
- Issue #807⁴⁴⁸³ - Create performance counters exposing overall AGAS statistics

⁴⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/831>

⁴⁴⁶² <https://github.com/STELLAR-GROUP/hpx/issues/830>

⁴⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/829>

⁴⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/828>

⁴⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/827>

⁴⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/826>

⁴⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/825>

⁴⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/824>

⁴⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/823>

⁴⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/822>

⁴⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/821>

⁴⁴⁷² <https://github.com/STELLAR-GROUP/hpx/issues/820>

⁴⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/819>

⁴⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/818>

⁴⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/817>

⁴⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/815>

⁴⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/814>

⁴⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/813>

⁴⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/811>

⁴⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/810>

⁴⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/809>

⁴⁴⁸² <https://github.com/STELLAR-GROUP/hpx/issues/808>

⁴⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/807>

- Issue #795⁴⁴⁸⁴ - Create timed make_ready_future
- Issue #794⁴⁴⁸⁵ - Create heterogeneous when_all/when_any/etc.
- Issue #721⁴⁴⁸⁶ - Make HPX usable for Xeon Phi
- Issue #694⁴⁴⁸⁷ - CMake should complain if you attempt to build an example without its dependencies
- Issue #692⁴⁴⁸⁸ - SLURM support broken
- Issue #683⁴⁴⁸⁹ - python/hpx/process.py imports epoll on all platforms
- Issue #619⁴⁴⁹⁰ - Automate the doc building process
- Issue #600⁴⁴⁹¹ - GTC performance broken
- Issue #577⁴⁴⁹² - Allow for zero copy serialization/networking
- Issue #551⁴⁴⁹³ - Change executable names to have debug postfix in Debug builds
- Issue #544⁴⁴⁹⁴ - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init
- Issue #534⁴⁴⁹⁵ - hpx::init should take functions by std::function and should accept all forms of hpx_main
- Issue #508⁴⁴⁹⁶ - FindPackage fails to set FOO_LIBRARY_DIR
- Issue #506⁴⁴⁹⁷ - Add cmake support to generate ini files for external applications
- Issue #470⁴⁴⁹⁸ - Changing build-type after configure does not update boost library names
- Issue #453⁴⁴⁹⁹ - Document hpx_run_tests.py
- Issue #445⁴⁵⁰⁰ - Significant performance mismatch between MPI and HPX in SMP for allgather example
- Issue #443⁴⁵⁰¹ - Make docs viewable from build directory
- Issue #421⁴⁵⁰² - Support multiple HPX instances per node in a batch environment like PBS or SLURM
- Issue #316⁴⁵⁰³ - Add message size limitation
- Issue #249⁴⁵⁰⁴ - Clean up locking code in big boot barrier
- Issue #136⁴⁵⁰⁵ - Persistent CMake variables need to be marked as cache variables

⁴⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/795>

⁴⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/794>

⁴⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/721>

⁴⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/694>

⁴⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/692>

⁴⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/683>

⁴⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/619>

⁴⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/600>

⁴⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/577>

⁴⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/551>

⁴⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/544>

⁴⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/534>

⁴⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/508>

⁴⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/506>

⁴⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/470>

⁴⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/453>

⁴⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/445>

⁴⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/443>

⁴⁵⁰² <https://github.com/STELLAR-GROUP/hpx/issues/421>

⁴⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/316>

⁴⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/249>

⁴⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/136>

2.10.19 HPX V0.9.6 (Jul 30, 2013)

We have had over 1200 commits since the last release and we have closed roughly 140 tickets (bugs, feature requests, etc.).

General changes

The major new features in this release are:

- We further consolidated the API exposed by *HPX*. We aligned our APIs as much as possible with the existing C++11 Standard⁴⁵⁰⁶ and related proposals to the C++ standardization committee (such as N3632⁴⁵⁰⁷ and N3857⁴⁵⁰⁸).
- We implemented a first version of a distributed AGAS service which essentially eliminates all explicit AGAS network traffic.
- We created a native ibverbs parcelport allowing to take advantage of the superior latency and bandwidth characteristics of Infiniband networks.
- We successfully ported *HPX* to the Xeon Phi platform.
- Support for the SLURM scheduling system was implemented.
- Major efforts have been dedicated to improving the performance counter framework, numerous new counters were implemented and new APIs were added.
- We added a modular parcel compression system allowing to improve bandwidth utilization (by reducing the overall size of the transferred data).
- We added a modular parcel coalescing system allowing to combine several parcels into larger messages. This reduces latencies introduced by the communication layer.
- Added an experimental executors API allowing to use different scheduling policies for different parts of the code. This API has been modelled after the Standards proposal N3562⁴⁵⁰⁹. This API is bound to change in the future, though.
- Added minimal security support for localities which is enforced on the parcelport level. This support is preliminary and experimental and might change in the future.
- We created a parcelport using low level MPI functions. This is in support of legacy applications which are to be gradually ported and to support platforms where MPI is the only available portable networking layer.
- We added a preliminary and experimental implementation of a tuple-space object which exposes an interface similar to such systems described in the literature (see for instance The Linda Coordination Language⁴⁵¹⁰).

⁴⁵⁰⁶ <http://www.open-std.org/jtc1/sc22/wg21>

⁴⁵⁰⁷ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3632.html>

⁴⁵⁰⁸ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf>

⁴⁵⁰⁹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf>

⁴⁵¹⁰ [https://en.wikipedia.org/wiki/Linda_\(coordination_language\)](https://en.wikipedia.org/wiki/Linda_(coordination_language))

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release. This is again a very long list of newly implemented features and fixed issues.

- Issue #806⁴⁵¹¹ - make (all) in examples folder does nothing
- Issue #805⁴⁵¹² - Adding the introduction and fixing DOCBOOK dependencies for Windows use
- Issue #804⁴⁵¹³ - Add stackless (non-suspendable) thread type
- Issue #803⁴⁵¹⁴ - Create proper serialization support functions for util::tuple
- Issue #800⁴⁵¹⁵ - Add possibility to disable array optimizations during serialization
- Issue #798⁴⁵¹⁶ - HPX_LIMIT does not work for local dataflow
- Issue #797⁴⁵¹⁷ - Create a parcelport which uses MPI
- Issue #796⁴⁵¹⁸ - Problem with Large Numbers of Threads
- Issue #793⁴⁵¹⁹ - Changing dataflow test case to hang consistently
- Issue #792⁴⁵²⁰ - CMake Error
- Issue #791⁴⁵²¹ - Problems with local::dataflow
- Issue #790⁴⁵²² - wait_for() doesn't compile
- Issue #789⁴⁵²³ - HPX with Intel compiler segfaults
- Issue #788⁴⁵²⁴ - Intel compiler support
- Issue #787⁴⁵²⁵ - Fixed SFINAEd specializations
- Issue #786⁴⁵²⁶ - Memory issues during benchmarking.
- Issue #785⁴⁵²⁷ - Create an API allowing to register external threads with HPX
- Issue #784⁴⁵²⁸ - util::plugin is throwing an error when a symbol is not found
- Issue #783⁴⁵²⁹ - How does hpx:bind work?
- Issue #782⁴⁵³⁰ - Added quotes around STRING REPLACE potentially empty arguments
- Issue #781⁴⁵³¹ - Make sure no exceptions propagate into the thread manager

⁴⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/806>

⁴⁵¹² <https://github.com/STELLAR-GROUP/hpx/issues/805>

⁴⁵¹³ <https://github.com/STELLAR-GROUP/hpx/issues/804>

⁴⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/803>

⁴⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/800>

⁴⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/798>

⁴⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/797>

⁴⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/796>

⁴⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/793>

⁴⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/792>

⁴⁵²¹ <https://github.com/STELLAR-GROUP/hpx/issues/791>

⁴⁵²² <https://github.com/STELLAR-GROUP/hpx/issues/790>

⁴⁵²³ <https://github.com/STELLAR-GROUP/hpx/issues/789>

⁴⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/788>

⁴⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/787>

⁴⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/786>

⁴⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/785>

⁴⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/784>

⁴⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/783>

⁴⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/782>

⁴⁵³¹ <https://github.com/STELLAR-GROUP/hpx/issues/781>

- Issue #780⁴⁵³² - Allow arithmetics performance counters to expand its parameters
- Issue #779⁴⁵³³ - Test case for 778
- Issue #778⁴⁵³⁴ - Swapping futures segfaults
- Issue #777⁴⁵³⁵ - hpx::lcos::details::when_xxx don't restore completion handlers
- Issue #776⁴⁵³⁶ - Compiler chokes on dataflow overload with launch policy
- Issue #775⁴⁵³⁷ - Runtime error with local dataflow (copying futures?)
- Issue #774⁴⁵³⁸ - Using local dataflow without explicit namespace
- Issue #773⁴⁵³⁹ - Local dataflow with unwrap: functor operators need to be const
- Issue #772⁴⁵⁴⁰ - Allow (remote) actions to return a future
- Issue #771⁴⁵⁴¹ - Setting HPX_LIMIT gives huge boost MPL errors
- Issue #770⁴⁵⁴² - Add launch policy to (local) dataflow
- Issue #769⁴⁵⁴³ - Make compile time configuration information available
- Issue #768⁴⁵⁴⁴ - Const correctness problem in local dataflow
- Issue #767⁴⁵⁴⁵ - Add launch policies to async
- Issue #766⁴⁵⁴⁶ - Mark data structures for optimized (array based) serialization
- Issue #765⁴⁵⁴⁷ - Align hpx::any with N3508: Any Library Proposal (Revision 2)
- Issue #764⁴⁵⁴⁸ - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations
- Issue #762⁴⁵⁴⁹ - added a human readable output for the ping pong example
- Issue #761⁴⁵⁵⁰ - Ambiguous typename when constructing derived component
- Issue #760⁴⁵⁵¹ - Simple components can not be derived
- Issue #759⁴⁵⁵² - make install doesn't give a complete install
- Issue #758⁴⁵⁵³ - Stack overflow when using locking_hook<>
- Issue #757⁴⁵⁵⁴ - copy paste error; unsupported function overloading

⁴⁵³² <https://github.com/STELLAR-GROUP/hpx/issues/780>

⁴⁵³³ <https://github.com/STELLAR-GROUP/hpx/issues/779>

⁴⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/778>

⁴⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/777>

⁴⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/776>

⁴⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/775>

⁴⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/774>

⁴⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/773>

⁴⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/772>

⁴⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/771>

⁴⁵⁴² <https://github.com/STELLAR-GROUP/hpx/issues/770>

⁴⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/769>

⁴⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/768>

⁴⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/767>

⁴⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/766>

⁴⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/765>

⁴⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/764>

⁴⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/762>

⁴⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/761>

⁴⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/760>

⁴⁵⁵² <https://github.com/STELLAR-GROUP/hpx/issues/759>

⁴⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/758>

⁴⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/757>

- Issue #756⁴⁵⁵⁵ - GTCX runtime issue in Gordon
- Issue #755⁴⁵⁵⁶ - Papi counters don't work with reset and evaluate API's
- Issue #753⁴⁵⁵⁷ - cmake bugfix and improved component action docs
- Issue #752⁴⁵⁵⁸ - hpx simple component docs
- Issue #750⁴⁵⁵⁹ - Add hpx::util::any
- Issue #749⁴⁵⁶⁰ - Thread phase counter is not reset
- Issue #748⁴⁵⁶¹ - Memory performance counter are not registered
- Issue #747⁴⁵⁶² - Create performance counters exposing arithmetic operations
- Issue #745⁴⁵⁶³ - apply_callback needs to invoke callback when applied locally
- Issue #744⁴⁵⁶⁴ - CMake fixes
- Issue #743⁴⁵⁶⁵ - Problem Building github version of HPX
- Issue #742⁴⁵⁶⁶ - Remove HPX_STD_BIND
- Issue #741⁴⁵⁶⁷ - assertion 'px != 0' failed: HPX(assertion_failure) for low numbers of OS threads
- Issue #739⁴⁵⁶⁸ - Performance counters do not count to the end of the program or evaluation
- Issue #738⁴⁵⁶⁹ - Dedicated AGAS server runs don't work; console ignores -a option.
- Issue #737⁴⁵⁷⁰ - Missing bind overloads
- Issue #736⁴⁵⁷¹ - Performance counter wildcards do not always work
- Issue #735⁴⁵⁷² - Create native ibverbs parcelport based on rdma operations
- Issue #734⁴⁵⁷³ - Threads stolen performance counter total is incorrect
- Issue #733⁴⁵⁷⁴ - Test benchmarks need to be checked and fixed
- Issue #732⁴⁵⁷⁵ - Build fails with Mac, using mac ports clang-3.3 on latest git branch
- Issue #731⁴⁵⁷⁶ - Add global start/stop API for performance counters
- Issue #730⁴⁵⁷⁷ - Performance counter values are apparently incorrect

⁴⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/756>

⁴⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/755>

⁴⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/753>

⁴⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/752>

⁴⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/750>

⁴⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/749>

4561 <https://github.com/STELLAR-GROUP/hpx/issues/748>4562 <https://github.com/STELLAR-GROUP/hpx/issues/747>4563 <https://github.com/STELLAR-GROUP/hpx/issues/745>4564 <https://github.com/STELLAR-GROUP/hpx/issues/744>4565 <https://github.com/STELLAR-GROUP/hpx/issues/743>4566 <https://github.com/STELLAR-GROUP/hpx/issues/742>4567 <https://github.com/STELLAR-GROUP/hpx/issues/741>4568 <https://github.com/STELLAR-GROUP/hpx/issues/739>4569 <https://github.com/STELLAR-GROUP/hpx/issues/738>4570 <https://github.com/STELLAR-GROUP/hpx/issues/737>4571 <https://github.com/STELLAR-GROUP/hpx/issues/736>4572 <https://github.com/STELLAR-GROUP/hpx/issues/735>4573 <https://github.com/STELLAR-GROUP/hpx/issues/734>4574 <https://github.com/STELLAR-GROUP/hpx/issues/733>4575 <https://github.com/STELLAR-GROUP/hpx/issues/732>4576 <https://github.com/STELLAR-GROUP/hpx/issues/731>4577 <https://github.com/STELLAR-GROUP/hpx/issues/730>

- Issue #729⁴⁵⁷⁸ - Unhandled switch
- Issue #728⁴⁵⁷⁹ - Serialization of hpx::util::function between two localities causes seg faults
- Issue #727⁴⁵⁸⁰ - Memory counters on Mac OS X
- Issue #725⁴⁵⁸¹ - Restore original thread priority on resume
- Issue #724⁴⁵⁸² - Performance benchmarks do not depend on main HPX libraries
- Issue #723⁴⁵⁸³ - [teletype]-hpx:nodes=``cat \$PBS_NODEFILE`` works; -hpx:nodefile=\$PBS_NODEFILE does not.[c++]
- Issue #722⁴⁵⁸⁴ - Fix binding const member functions as actions
- Issue #719⁴⁵⁸⁵ - Create performance counter exposing compression ratio
- Issue #718⁴⁵⁸⁶ - Add possibility to compress parcel data
- Issue #717⁴⁵⁸⁷ - strip_credit_from_gid has misleading semantics
- Issue #716⁴⁵⁸⁸ - Non-option arguments to programs run using pbsdsh must be before --hpx:nodes, contrary to directions
- Issue #715⁴⁵⁸⁹ - Re-thrown exceptions should retain the original call site
- Issue #714⁴⁵⁹⁰ - failed assertion in debug mode
- Issue #713⁴⁵⁹¹ - Add performance counters monitoring connection caches
- Issue #712⁴⁵⁹² - Adjust parcel related performance counters to be connection type specific
- Issue #711⁴⁵⁹³ - configuration failure
- Issue #710⁴⁵⁹⁴ - Error “timed out while trying to find room in the connection cache” when trying to start multiple localities on a single computer
- Issue #709⁴⁵⁹⁵ - Add new thread state ‘staged’ referring to task descriptions
- Issue #708⁴⁵⁹⁶ - Detect/mitigate bad non-system installs of GCC on Redhat systems
- Issue #707⁴⁵⁹⁷ - Many examples do not link with Git HEAD version
- Issue #706⁴⁵⁹⁸ - hpx::init removes portions of non-option command line arguments before last = sign
- Issue #705⁴⁵⁹⁹ - Create rolling average and median aggregating performance counters

⁴⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/729>

⁴⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/728>

⁴⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/727>

⁴⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/725>

⁴⁵⁸² <https://github.com/STELLAR-GROUP/hpx/issues/724>

⁴⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/723>

⁴⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/722>

⁴⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/719>

⁴⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/718>

⁴⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/717>

⁴⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/716>

⁴⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/715>

⁴⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/714>

⁴⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/713>

⁴⁵⁹² <https://github.com/STELLAR-GROUP/hpx/issues/712>

⁴⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/711>

⁴⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/710>

⁴⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/709>

⁴⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/708>

⁴⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/707>

⁴⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/706>

⁴⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/705>

- Issue #704⁴⁶⁰⁰ - Create performance counter to expose thread queue waiting time
- Issue #703⁴⁶⁰¹ - Add support to HPX build system to find libcrctool.a and related headers
- Issue #699⁴⁶⁰² - Generalize instrumentation support
- Issue #698⁴⁶⁰³ - compilation failure with hwloc absent
- Issue #697⁴⁶⁰⁴ - Performance counter counts should be zero indexed
- Issue #696⁴⁶⁰⁵ - Distributed problem
- Issue #695⁴⁶⁰⁶ - Bad perf counter time printed
- Issue #693⁴⁶⁰⁷ - --help doesn't print component specific command line options
- Issue #692⁴⁶⁰⁸ - SLURM support broken
- Issue #691⁴⁶⁰⁹ - exception while executing any application linked with hwloc
- Issue #690⁴⁶¹⁰ - thread_id_test and thread_launcher_test failing
- Issue #689⁴⁶¹¹ - Make the buildbots use hwloc
- Issue #687⁴⁶¹² - compilation error fix (hwloc_topology)
- Issue #686⁴⁶¹³ - Linker Error for Applications
- Issue #684⁴⁶¹⁴ - Pinning of service thread fails when number of worker threads equals the number of cores
- Issue #682⁴⁶¹⁵ - Add performance counters exposing number of stolen threads
- Issue #681⁴⁶¹⁶ - Add apply_continue for asynchronous chaining of actions
- Issue #679⁴⁶¹⁷ - Remove obsolete async_callback API functions
- Issue #678⁴⁶¹⁸ - Add new API for setting/triggering LCOs
- Issue #677⁴⁶¹⁹ - Add async_continue for true continuation style actions
- Issue #676⁴⁶²⁰ - Buildbot for gcc 4.4 broken
- Issue #675⁴⁶²¹ - Partial preprocessing broken
- Issue #674⁴⁶²² - HPX segfaults when built with gcc 4.7

⁴⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/704>

⁴⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/703>

⁴⁶⁰² <https://github.com/STELLAR-GROUP/hpx/issues/699>

⁴⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/698>

⁴⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/697>

⁴⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/696>

4606 <https://github.com/STELLAR-GROUP/hpx/issues/695>4607 <https://github.com/STELLAR-GROUP/hpx/issues/693>4608 <https://github.com/STELLAR-GROUP/hpx/issues/692>4609 <https://github.com/STELLAR-GROUP/hpx/issues/691>4610 <https://github.com/STELLAR-GROUP/hpx/issues/690>4611 <https://github.com/STELLAR-GROUP/hpx/issues/689>4612 <https://github.com/STELLAR-GROUP/hpx/issues/687>4613 <https://github.com/STELLAR-GROUP/hpx/issues/686>4614 <https://github.com/STELLAR-GROUP/hpx/issues/684>4615 <https://github.com/STELLAR-GROUP/hpx/issues/682>4616 <https://github.com/STELLAR-GROUP/hpx/issues/681>4617 <https://github.com/STELLAR-GROUP/hpx/issues/679>4618 <https://github.com/STELLAR-GROUP/hpx/issues/678>4619 <https://github.com/STELLAR-GROUP/hpx/issues/677>4620 <https://github.com/STELLAR-GROUP/hpx/issues/676>4621 <https://github.com/STELLAR-GROUP/hpx/issues/675>4622 <https://github.com/STELLAR-GROUP/hpx/issues/674>

- Issue #673⁴⁶²³ - use_guard_pages has inconsistent preprocessor guards
- Issue #672⁴⁶²⁴ - External build breaks if library path has spaces
- Issue #671⁴⁶²⁵ - release tarballs are tarbombs
- Issue #670⁴⁶²⁶ - CMake won't find Boost headers in layout=versioned install
- Issue #669⁴⁶²⁷ - Links in docs to source files broken if not installed
- Issue #667⁴⁶²⁸ - Not reading ini file properly
- Issue #664⁴⁶²⁹ - Adapt new meanings of ‘const’ and ‘mutable’
- Issue #661⁴⁶³⁰ - Implement BTL Parcel port
- Issue #655⁴⁶³¹ - Make HPX work with the “decltype” result_of
- Issue #647⁴⁶³² - documentation for specifying the number of high priority threads --hpx:high-priority-threads
- Issue #643⁴⁶³³ - Error parsing host file
- Issue #642⁴⁶³⁴ - HWLoc issue with TAU
- Issue #639⁴⁶³⁵ - Logging potentially suspends a running thread
- Issue #634⁴⁶³⁶ - Improve error reporting from parcel layer
- Issue #627⁴⁶³⁷ - Add tests for async and apply overloads that accept regular C++ functions
- Issue #626⁴⁶³⁸ - hpx/future.hpp header
- Issue #601⁴⁶³⁹ - Intel support
- Issue #557⁴⁶⁴⁰ - Remove action codes
- Issue #531⁴⁶⁴¹ - AGAS request and response classes should use switch statements
- Issue #529⁴⁶⁴² - Investigate the state of hwloc support
- Issue #526⁴⁶⁴³ - Make HPX aware of hyper-threading
- Issue #518⁴⁶⁴⁴ - Create facilities allowing to use plain arrays as action arguments
- Issue #473⁴⁶⁴⁵ - hwloc thread binding is broken on CPUs with hyperthreading

⁴⁶²³ <https://github.com/STELLAR-GROUP/hpx/issues/673>

⁴⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/672>

⁴⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/671>

⁴⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/670>

⁴⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/669>

⁴⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/667>

⁴⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/664>

⁴⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/661>

⁴⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/655>

⁴⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/647>

⁴⁶³³ <https://github.com/STELLAR-GROUP/hpx/issues/643>

⁴⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/642>

⁴⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/639>

⁴⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/634>

⁴⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/627>

⁴⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/626>

⁴⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/601>

⁴⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/557>

⁴⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/531>

⁴⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/529>

⁴⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/526>

⁴⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/518>

⁴⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/473>

- Issue #383⁴⁶⁴⁶ - Change result type detection for hpx::util::bind to use result_of protocol
- Issue #341⁴⁶⁴⁷ - Consolidate route code
- Issue #219⁴⁶⁴⁸ - Only copy arguments into actions once
- Issue #177⁴⁶⁴⁹ - Implement distributed AGAS
- Issue #43⁴⁶⁵⁰ - Support for Darwin (Xcode + Clang)

2.10.20 HPX V0.9.5 (Jan 16, 2013)

We have had over 1000 commits since the last release and we have closed roughly 150 tickets (bugs, feature requests, etc.).

General changes

This release is continuing along the lines of code and API consolidation, and overall usability improvements. We dedicated much attention to performance and we were able to significantly improve the threading and networking subsystems.

We successfully ported *HPX* to the Android platform. *HPX* applications now not only can run on mobile devices, but we support heterogeneous applications running across architecture boundaries. At the Supercomputing Conference 2012 we demonstrated connecting Android tablets to simulations running on a Linux cluster. The Android tablet was used to query performance counters from the Linux simulation and to steer its parameters.

We successfully ported *HPX* to Mac OSX (using the Clang compiler). Thanks to Pyry Jahkola for contributing the corresponding patches. Please see the section macos_installation for more details.

We made a special effort to make *HPX* usable in highly concurrent use cases. Many of the *HPX* API functions which possibly take longer than 100 microseconds to execute now can be invoked asynchronously. We added uniform support for composing futures which simplifies to write asynchronous code. *HPX* actions (function objects encapsulating possibly concurrent remote function invocations) are now well integrated with all other API facilities such like `hpx::bind`.

All of the API has been aligned as much as possible with established paradigms. *HPX* now mirrors many of the facilities as defined in the C++11 Standard, such as `hpx::thread`, `hpx::function`, `hpx::future`, etc.

A lot of work has been put into improving the documentation. Many of the API functions are documented now, concepts are explained in detail, and examples are better described than before. The new documentation index enables finding information with lesser effort.

This is the first release of *HPX* we perform after the move to [Github](#)⁴⁶⁵¹ 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](#)⁴⁶⁵²).

⁴⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/383>

⁴⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/341>

⁴⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/219>

⁴⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/177>

⁴⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/43>

⁴⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/>

⁴⁶⁵² https://www.boost.org/LICENSE_1_0.txt

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 #666⁴⁶⁵³ - Segfault on calling hpx::finalize twice
- Issue #665⁴⁶⁵⁴ - Adding declaration num_of_cores
- Issue #662⁴⁶⁵⁵ - pkgconfig is building wrong
- Issue #660⁴⁶⁵⁶ - Need uninterrupt function
- Issue #659⁴⁶⁵⁷ - Move our logging library into a different namespace
- Issue #658⁴⁶⁵⁸ - Dynamic performance counter types are broken
- Issue #657⁴⁶⁵⁹ - HPX v0.9.5 (RC1) hello_world example segfaulting
- Issue #656⁴⁶⁶⁰ - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- Issue #654⁴⁶⁶¹ - Integrate the Boost auto_index tool with documentation
- Issue #653⁴⁶⁶² - Make HPX build on OS X + Clang + libc++
- Issue #651⁴⁶⁶³ - Add fine-grained control for thread pinning
- Issue #650⁴⁶⁶⁴ - Command line no error message when using -hpx:(anything)
- Issue #645⁴⁶⁶⁵ - Command line aliases don't work in [teletype]` `@file` `[c++]
- Issue #644⁴⁶⁶⁶ - Terminated threads are not always properly cleaned up
- Issue #640⁴⁶⁶⁷ - future_data<T>::set_on_completed_ used without locks
- Issue #638⁴⁶⁶⁸ - hpx build with intel compilers fails on linux
- Issue #637⁴⁶⁶⁹ - --copy-dt-needed-entries breaks with gold
- Issue #635⁴⁶⁷⁰ - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- Issue #633⁴⁶⁷¹ - Re-add examples to final 0.9.5 release
- Issue #632⁴⁶⁷² - Example thread_aware_timer is broken
- Issue #631⁴⁶⁷³ - FFT application throws error in parcellayer

⁴⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/666>

⁴⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/665>

⁴⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/662>

⁴⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/660>

⁴⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/659>

⁴⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/658>

⁴⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/657>

⁴⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/656>

⁴⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/654>

⁴⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/653>

⁴⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/651>

⁴⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/650>

⁴⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/645>

⁴⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/644>

⁴⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/640>

⁴⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/638>

⁴⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/637>

⁴⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/635>

⁴⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/633>

⁴⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/632>

⁴⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/631>

- Issue #630⁴⁶⁷⁴ - Event synchronization example is broken
- Issue #629⁴⁶⁷⁵ - Waiting on futures hangs
- Issue #628⁴⁶⁷⁶ - Add an HPX_ALWAYS_ASSERT macro
- Issue #625⁴⁶⁷⁷ - Port coroutines context switch benchmark
- Issue #621⁴⁶⁷⁸ - New INI section for stack sizes
- Issue #618⁴⁶⁷⁹ - pkg_config support does not work with a HPX debug build
- Issue #617⁴⁶⁸⁰ - hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: ‘get_thread_id’ was not declared in this scope
- Issue #616⁴⁶⁸¹ - Change wait_xxx not to use locking
- Issue #615⁴⁶⁸² - Revert visibility ‘fix’ (fb0b6b8245dad1127b0c25ebafd9386b3945cca9)
- Issue #614⁴⁶⁸³ - Fix Dataflow linker error
- Issue #613⁴⁶⁸⁴ - find_here should throw an exception on failure
- Issue #612⁴⁶⁸⁵ - Thread phase doesn’t show up in debug mode
- Issue #611⁴⁶⁸⁶ - Make stack guard pages configurable at runtime (initialization time)
- Issue #610⁴⁶⁸⁷ - Co-Locate Components
- Issue #609⁴⁶⁸⁸ - future_overhead
- Issue #608⁴⁶⁸⁹ - --hpx:list-counter-infos problem
- Issue #607⁴⁶⁹⁰ - Update Boost.Context based backend for coroutines
- Issue #606⁴⁶⁹¹ - 1d_wave_equation is not working
- Issue #605⁴⁶⁹² - Any C++ function that has serializable arguments and a serializable return type should be remotable
- Issue #604⁴⁶⁹³ - Connecting localities isn’t working anymore
- Issue #603⁴⁶⁹⁴ - Do not verify any ini entries read from a file
- Issue #602⁴⁶⁹⁵ - Rename argument_size to type_size/ added implementation to get parcel size

⁴⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/630>

⁴⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/629>

⁴⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/628>

⁴⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/625>

⁴⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/621>

⁴⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/618>

⁴⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/617>

⁴⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/616>

⁴⁶⁸² <https://github.com/STELLAR-GROUP/hpx/issues/615>

⁴⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/614>

⁴⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/613>

4685 <https://github.com/STELLAR-GROUP/hpx/issues/612>4686 <https://github.com/STELLAR-GROUP/hpx/issues/611>4687 <https://github.com/STELLAR-GROUP/hpx/issues/610>4688 <https://github.com/STELLAR-GROUP/hpx/issues/609>4689 <https://github.com/STELLAR-GROUP/hpx/issues/608>4690 <https://github.com/STELLAR-GROUP/hpx/issues/607>4691 <https://github.com/STELLAR-GROUP/hpx/issues/606>4692 <https://github.com/STELLAR-GROUP/hpx/issues/605>4693 <https://github.com/STELLAR-GROUP/hpx/issues/604>4694 <https://github.com/STELLAR-GROUP/hpx/issues/603>4695 <https://github.com/STELLAR-GROUP/hpx/issues/602>

- Issue #599⁴⁶⁹⁶ - Enable locality specific command line options
- Issue #598⁴⁶⁹⁷ - Need an API that accesses the performance counter reporting the system uptime
- Issue #597⁴⁶⁹⁸ - compiling on ranger
- Issue #595⁴⁶⁹⁹ - I need a place to store data in a thread self pointer
- Issue #594⁴⁷⁰⁰ - 32/64 interoperability
- Issue #593⁴⁷⁰¹ - Warn if logging is disabled at compile time but requested at runtime
- Issue #592⁴⁷⁰² - Add optional argument value to --hpx:list-counters and --hpx:list-counter-infos
- Issue #591⁴⁷⁰³ - Allow for wildcards in performance counter names specified with --hpx:print-counter
- Issue #590⁴⁷⁰⁴ - Local promise semantic differences
- Issue #589⁴⁷⁰⁵ - Create API to query performance counter names
- Issue #587⁴⁷⁰⁶ - Add get_num_localities and get_num_threads to AGAS API
- Issue #586⁴⁷⁰⁷ - Adjust local AGAS cache size based on number of localities
- Issue #585⁴⁷⁰⁸ - Error while using counters in HPX
- Issue #584⁴⁷⁰⁹ - counting argument size of actions, initial pass.
- Issue #581⁴⁷¹⁰ - Remove RemoteResult template parameter for future<>
- Issue #580⁴⁷¹¹ - Add possibility to hook into actions
- Issue #578⁴⁷¹² - Use angle brackets in HPX error dumps
- Issue #576⁴⁷¹³ - Exception incorrectly thrown when --help is used
- Issue #575⁴⁷¹⁴ - HPX(bad_component_type) with gcc 4.7.2 and boost 1.51
- Issue #574⁴⁷¹⁵ - --hpx:connect command line parameter not working correctly
- Issue #571⁴⁷¹⁶ - hpx::wait () (callback version) should pass the future to the callback function
- Issue #570⁴⁷¹⁷ - hpx::wait should operate on boost::arrays and std::lists
- Issue #569⁴⁷¹⁸ - Add a logging sink for Android

⁴⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/599>

⁴⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/598>

⁴⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/597>

⁴⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/595>

⁴⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/594>

⁴⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/593>

⁴⁷⁰² <https://github.com/STELLAR-GROUP/hpx/issues/592>

⁴⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/591>

⁴⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/590>

⁴⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/589>

⁴⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/587>

⁴⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/586>

⁴⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/585>

⁴⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/584>

⁴⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/581>

⁴⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/580>

⁴⁷¹² <https://github.com/STELLAR-GROUP/hpx/issues/578>

⁴⁷¹³ <https://github.com/STELLAR-GROUP/hpx/issues/576>

⁴⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/575>

⁴⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/574>

⁴⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/571>

⁴⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/570>

⁴⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/569>

- Issue #568⁴⁷¹⁹ - 2-argument version of HPX_DEFINE_COMPONENT_ACTION
- Issue #567⁴⁷²⁰ - Connecting to a running HPX application works only once
- Issue #565⁴⁷²¹ - HPX doesn't shutdown properly
- Issue #564⁴⁷²² - Partial preprocessing of new component creation interface
- Issue #563⁴⁷²³ - Add hpx::start/hpx::stop to avoid blocking main thread
- Issue #562⁴⁷²⁴ - All command line arguments swallowed by hpx
- Issue #561⁴⁷²⁵ - Boost.Tuple is not move aware
- Issue #558⁴⁷²⁶ - boost::shared_ptr<> style semantics/syntax for client classes
- Issue #556⁴⁷²⁷ - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
- Issue #555⁴⁷²⁸ - BOOST_FORCEINLINE does not name a type
- Issue #554⁴⁷²⁹ - Possible race condition in thread get_id()
- Issue #552⁴⁷³⁰ - Move enable client_base
- Issue #550⁴⁷³¹ - Add stack size category 'huge'
- Issue #549⁴⁷³² - ShenEOS run seg-faults on single or distributed runs
- Issue #545⁴⁷³³ - AUTOLOB broken for add_hpx_component
- Issue #542⁴⁷³⁴ - FindHPX_HDF5 still searches multiple times
- Issue #541⁴⁷³⁵ - Quotes around application name in hpx::init
- Issue #539⁴⁷³⁶ - Race condition occurring with new lightweight threads
- Issue #535⁴⁷³⁷ - hpx_run_tests.py exits with no error code when tests are missing
- Issue #530⁴⁷³⁸ - Thread description(<unknown>) in logs
- Issue #523⁴⁷³⁹ - Make thread objects more lightweight
- Issue #521⁴⁷⁴⁰ - hpx::error_code is not usable for lightweight error handling
- Issue #520⁴⁷⁴¹ - Add full user environment to HPX logs

⁴⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/568>

⁴⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/567>

⁴⁷²¹ <https://github.com/STELLAR-GROUP/hpx/issues/565>

⁴⁷²² <https://github.com/STELLAR-GROUP/hpx/issues/564>

⁴⁷²³ <https://github.com/STELLAR-GROUP/hpx/issues/563>

⁴⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/562>

⁴⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/561>

⁴⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/558>

⁴⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/556>

⁴⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/555>

⁴⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/554>

⁴⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/552>

⁴⁷³¹ <https://github.com/STELLAR-GROUP/hpx/issues/550>

⁴⁷³² <https://github.com/STELLAR-GROUP/hpx/issues/549>

⁴⁷³³ <https://github.com/STELLAR-GROUP/hpx/issues/545>

⁴⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/542>

⁴⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/541>

⁴⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/539>

⁴⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/535>

⁴⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/530>

⁴⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/523>

⁴⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/521>

⁴⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/520>

- Issue #519⁴⁷⁴² - Build succeeds, running fails
- Issue #517⁴⁷⁴³ - Add a guard page to linux coroutine stacks
- Issue #516⁴⁷⁴⁴ - hpx::thread::detach suspends while holding locks, leads to hang in debug
- Issue #514⁴⁷⁴⁵ - Preprocessed headers for <hpx/apply.hpp> don't compile
- Issue #513⁴⁷⁴⁶ - Buildbot configuration problem
- Issue #512⁴⁷⁴⁷ - Implement action based stack size customization
- Issue #511⁴⁷⁴⁸ - Move action priority into a separate type trait
- Issue #510⁴⁷⁴⁹ - trunk broken
- Issue #507⁴⁷⁵⁰ - no matching function for call to boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(hpx::threads::topology*)
- Issue #505⁴⁷⁵¹ - undefined_symbol regression test currently failing
- Issue #502⁴⁷⁵² - Adding OpenCL and OCLM support to HPX for Windows and Linux
- Issue #501⁴⁷⁵³ - find_package(HPX) sets cmake output variables
- Issue #500⁴⁷⁵⁴ - wait_any/wait_all are badly named
- Issue #499⁴⁷⁵⁵ - Add support for disabling pbs support in pbs runs
- Issue #498⁴⁷⁵⁶ - Error during no-cache runs
- Issue #496⁴⁷⁵⁷ - Add partial preprocessing support to cmake
- Issue #495⁴⁷⁵⁸ - Support HPX modules exporting startup/shutdown functions only
- Issue #494⁴⁷⁵⁹ - Allow modules to specify when to run startup/shutdown functions
- Issue #493⁴⁷⁶⁰ - Avoid constructing a string in make_success_code
- Issue #492⁴⁷⁶¹ - Performance counter creation is no longer synchronized at startup
- Issue #491⁴⁷⁶² - Performance counter creation is no longer synchronized at startup
- Issue #490⁴⁷⁶³ - Sheneos on_completed_bulk seg fault in distributed
- Issue #489⁴⁷⁶⁴ - compiling issue with g++44

⁴⁷⁴² <https://github.com/STELLAR-GROUP/hpx/issues/519>

⁴⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/517>

⁴⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/516>

⁴⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/514>

⁴⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/513>

⁴⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/512>

⁴⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/511>

⁴⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/510>

⁴⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/507>

⁴⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/505>

⁴⁷⁵² <https://github.com/STELLAR-GROUP/hpx/issues/502>

⁴⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/501>

⁴⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/500>

⁴⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/499>

⁴⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/498>

⁴⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/496>

⁴⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/495>

⁴⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/494>

⁴⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/493>

⁴⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/492>

⁴⁷⁶² <https://github.com/STELLAR-GROUP/hpx/issues/491>

⁴⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/490>

⁴⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/489>

- Issue #488⁴⁷⁶⁵ - Adding OpenCL and OCLM support to HPX for the MSVC platform
- Issue #487⁴⁷⁶⁶ - FindHPX.cmake problems
- Issue #485⁴⁷⁶⁷ - Change distributing_factory and binpacking_factory to use bulk creation
- Issue #484⁴⁷⁶⁸ - Change HPX_DONT_USE_PREPROCESSED_FILES to HPX_USE_PREPROCESSED_FILES
- Issue #483⁴⁷⁶⁹ - Memory counter for Windows
- Issue #479⁴⁷⁷⁰ - strange errors appear when requesting performance counters on multiple nodes
- Issue #477⁴⁷⁷¹ - Create (global) timer for multi-threaded measurements
- Issue #472⁴⁷⁷² - Add partial preprocessing using Wave
- Issue #471⁴⁷⁷³ - Segfault stack traces don't show up in release
- Issue #468⁴⁷⁷⁴ - External projects need to link with internal components
- Issue #462⁴⁷⁷⁵ - Startup/shutdown functions are called more than once
- Issue #458⁴⁷⁷⁶ - Consolidate hpx::util::high_resolution_timer and hpx::util::high_resolution_clock
- Issue #457⁴⁷⁷⁷ - index out of bounds in allgather_and_gate on 4 cores or more
- Issue #448⁴⁷⁷⁸ - Make HPX compile with clang
- Issue #447⁴⁷⁷⁹ - 'make tests' should execute tests on local installation
- Issue #446⁴⁷⁸⁰ - Remove SVN-related code from the codebase
- Issue #444⁴⁷⁸¹ - race condition in smp
- Issue #441⁴⁷⁸² - Patched Boost.Serialization headers should only be installed if needed
- Issue #439⁴⁷⁸³ - Components using HPX_REGISTER_STARTUP_MODULE fail to compile with MSVC
- Issue #436⁴⁷⁸⁴ - Verify that no locks are being held while threads are suspended
- Issue #435⁴⁷⁸⁵ - Installing HPX should not clobber existing Boost installation
- Issue #434⁴⁷⁸⁶ - Logging external component failed (Boost 1.50)
- Issue #433⁴⁷⁸⁷ - Runtime crash when building all examples

⁴⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/488>

⁴⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/487>

⁴⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/485>

⁴⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/484>

⁴⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/483>

⁴⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/479>

⁴⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/477>

⁴⁷⁷² <https://github.com/STELLAR-GROUP/hpx/issues/472>

⁴⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/471>

⁴⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/468>

⁴⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/462>

⁴⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/458>

⁴⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/457>

⁴⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/448>

⁴⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/447>

⁴⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/446>

⁴⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/444>

⁴⁷⁸² <https://github.com/STELLAR-GROUP/hpx/issues/441>

⁴⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/439>

⁴⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/436>

⁴⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/435>

⁴⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/434>

⁴⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/433>

- Issue #432⁴⁷⁸⁸ - Dataflow hangs on 512 cores/64 nodes
- Issue #430⁴⁷⁸⁹ - Problem with distributing factory
- Issue #424⁴⁷⁹⁰ - File paths referring to XSL-files need to be properly escaped
- Issue #417⁴⁷⁹¹ - Make dataflow LCOs work out of the box by using partial preprocessing
- Issue #413⁴⁷⁹² - hpx_svnversion.py fails on Windows
- Issue #412⁴⁷⁹³ - Make hpx::error_code equivalent to hpx::exception
- Issue #398⁴⁷⁹⁴ - HPX clobbers out-of-tree application specific CMake variables (specifically CMAKE_BUILD_TYPE)
- Issue #394⁴⁷⁹⁵ - Remove code generating random port numbers for network
- Issue #378⁴⁷⁹⁶ - ShenEOS scaling issues
- Issue #354⁴⁷⁹⁷ - Create a coroutines wrapper for Boost.Context
- Issue #349⁴⁷⁹⁸ - Commandline option --localities=N/-lN should be necessary only on AGAS locality
- Issue #334⁴⁷⁹⁹ - Add auto_index support to cmake based documentation toolchain
- Issue #318⁴⁸⁰⁰ - Network benchmarks
- Issue #317⁴⁸⁰¹ - Implement network performance counters
- Issue #310⁴⁸⁰² - Duplicate logging entries
- Issue #230⁴⁸⁰³ - Add compile time option to disable thread debugging info
- Issue #171⁴⁸⁰⁴ - Add an INI option to turn off deadlock detection independently of logging
- Issue #170⁴⁸⁰⁵ - OSHL internal counters are incorrect
- Issue #103⁴⁸⁰⁶ - Better diagnostics for multiple component/action registrations under the same name
- Issue #48⁴⁸⁰⁷ - Support for Darwin (Xcode + Clang)
- Issue #21⁴⁸⁰⁸ - Build fails with GCC 4.6

4788 <https://github.com/STELLAR-GROUP/hpx/issues/432>

4789 <https://github.com/STELLAR-GROUP/hpx/issues/430>

4790 <https://github.com/STELLAR-GROUP/hpx/issues/424>

4791 <https://github.com/STELLAR-GROUP/hpx/issues/417>

4792 <https://github.com/STELLAR-GROUP/hpx/issues/413>

4793 <https://github.com/STELLAR-GROUP/hpx/issues/412>

4794 <https://github.com/STELLAR-GROUP/hpx/issues/398>

4795 <https://github.com/STELLAR-GROUP/hpx/issues/394>

4796 <https://github.com/STELLAR-GROUP/hpx/issues/378>

4797 <https://github.com/STELLAR-GROUP/hpx/issues/354>

4798 <https://github.com/STELLAR-GROUP/hpx/issues/349>

4799 <https://github.com/STELLAR-GROUP/hpx/issues/334>

4800 <https://github.com/STELLAR-GROUP/hpx/issues/318>

4801 <https://github.com/STELLAR-GROUP/hpx/issues/317>

4802 <https://github.com/STELLAR-GROUP/hpx/issues/310>

4803 <https://github.com/STELLAR-GROUP/hpx/issues/230>

4804 <https://github.com/STELLAR-GROUP/hpx/issues/171>

4805 <https://github.com/STELLAR-GROUP/hpx/issues/170>

4806 <https://github.com/STELLAR-GROUP/hpx/issues/103>

4807 <https://github.com/STELLAR-GROUP/hpx/issues/48>

4808 <https://github.com/STELLAR-GROUP/hpx/issues/21>

2.10.21 HPX V0.9.0 (Jul 5, 2012)

We have had roughly 800 commits since the last release and we have closed approximately 80 tickets (bugs, feature requests, etc.).

General changes

- Significant improvements made to the usability of *HPX* in large-scale, distributed environments.
- Renamed `hpx::lcos::packaged_task` to `hpx::lcos::packaged_action` to reflect the semantic differences to a `packaged_task` as defined by the C++11 Standard⁴⁸⁰⁹.
- *HPX* now exposes `hpx::thread` which is compliant to the C++11 `std::thread` type except that it (purely locally) represents an *HPX* thread. This new type does not expose any of the remote capabilities of the underlying *HPX*-thread implementation.
- The type `hpx::lcos::future` is now compliant to the C++11 `std::future<>` type. This type can be used to synchronize both, local and remote operations. In both cases the control flow will ‘return’ to the future in order to trigger any continuation.
- The types `hpx::lcos::local::promise` and `hpx::lcos::local::packaged_task` are now compliant to the C++11 `std::promise<>` and `std::packaged_task<>` types. These can be used to create a future representing local work only. Use the types `hpx::lcos::promise` and `hpx::lcos::packaged_action` to wrap any (possibly remote) action into a future.
- `hpx::thread` and `hpx::lcos::future` are now cancelable.
- Added support for sequential and logic composition of `hpx::lcos::futures`. The member function `hpx::lcos::future::when` permits futures to be sequentially composed. The helper functions `hpx::wait_all`, `hpx::wait_any`, and `hpx::wait_n` can be used to wait for more than one future at a time.
- *HPX* now exposes `hpx::apply` and `hpx::async` as the preferred way of creating (or invoking) any deferred work. These functions are usable with various types of functions, function objects, and actions and provide a uniform way to spawn deferred tasks.
- *HPX* now utilizes `hpx::util::bind` to (partially) bind local functions and function objects, and also actions. Remote bound actions can have placeholders as well.
- *HPX* continuations are now fully polymorphic. The class `hpx::actions::forwarding_continuation` is an example of how the user can write their 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();
}
```

(continues on next page)

⁴⁸⁰⁹ <http://www.open-std.org/jtc1/sc22/wg21>

(continued from previous page)

```

}

int main()
{
    return hpx::init();
}

```

This removes the immediate dependency on the [Boost.Program Options⁴⁸¹⁰](#) 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 Options⁴⁸¹¹](#), the minimal program can be written as:

```

#include <hpx/hpx_init.hpp>

// The arguments for hpx_main can be left off, which very similar to the
// behavior of ``main()`` as defined by C++.
int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);
}

```

- Added performance counters exposing the number of component instances which are alive on a given locality.
- Added performance counters exposing then number of messages sent and received, the number of parcels sent and received, the number of bytes sent and received, the overall time required to send and receive data, and the overall time required to serialize and deserialize the data.
- Added a new component: `hpx::components::binpacking_factory` which is equivalent to the existing `hpx::components::distributing_factory` component, except that it equalizes the overall population of the components to create. It exposes two factory methods, one based on the number of existing instances of the component type to create, and one based on an arbitrary performance counter which will be queried for all relevant localities.
- Added API functions allowing to access elements of the diagnostic information embedded in the given exception: `hpx::get_locality_id`, `hpx::get_host_name`, `hpx::get_process_id`, `hpx::get_function_name`, `hpx::get_file_name`, `hpx::get_line_number`, `hpx::get_os_thread`, `hpx::get_thread_id`, and `hpx::get_thread_description`.

⁴⁸¹⁰ https://www.boost.org/doc/html/program_options.html

⁴⁸¹¹ https://www.boost.org/doc/html/program_options.html

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #71⁴⁸¹² - GIDs that are not serialized via `handle_gid<>` should raise an exception
- Issue #105⁴⁸¹³ - Allow for `hpx::util::functions` to be registered in the AGAS symbolic namespace
- Issue #107⁴⁸¹⁴ - Nasty threadmanger race condition (reproducible in `sheneos_test`)
- Issue #108⁴⁸¹⁵ - Add millisecond resolution to *HPX* logs on Linux
- Issue #110⁴⁸¹⁶ - Shutdown hang in distributed with release build
- Issue #116⁴⁸¹⁷ - Don't use TSS for the applier and runtime pointers
- Issue #162⁴⁸¹⁸ - Move local synchronous execution shortcut from `hpx::function` to the applier
- Issue #172⁴⁸¹⁹ - Cache sources in CMake and check if they change manually
- Issue #178⁴⁸²⁰ - Add an INI option to turn off ranged-based AGAS caching
- Issue #187⁴⁸²¹ - Support for disabling performance counter deployment
- Issue #202⁴⁸²² - Support for sending performance counter data to a specific file
- Issue #218⁴⁸²³ - `boost.coroutines` allows different stack sizes, but stack pool is unaware of this
- Issue #231⁴⁸²⁴ - Implement movable `boost::bind`
- Issue #232⁴⁸²⁵ - Implement movable `boost::function`
- Issue #236⁴⁸²⁶ - Allow binding `hpx::util::function` to actions
- Issue #239⁴⁸²⁷ - Replace `hpx::function` with `hpx::util::function`
- Issue #240⁴⁸²⁸ - Can't specify `RemoteResult` with `lcos::async`
- Issue #242⁴⁸²⁹ - `REGISTER_TEMPLATE` support for plain actions
- Issue #243⁴⁸³⁰ - `handle_gid<>` support for `hpx::util::function`
- Issue #245⁴⁸³¹ - `*_c_cache` code throws an exception if the queried GID is not in the local cache
- Issue #246⁴⁸³² - Undefined references in dataflow/adaptive1d example

⁴⁸¹² <https://github.com/STELLAR-GROUP/hpx/issues/71>

⁴⁸¹³ <https://github.com/STELLAR-GROUP/hpx/issues/105>

⁴⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/107>

⁴⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/108>

⁴⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/110>

⁴⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/116>

⁴⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/162>

⁴⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/172>

⁴⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/178>

4821 <https://github.com/STELLAR-GROUP/hpx/issues/187>4822 <https://github.com/STELLAR-GROUP/hpx/issues/202>4823 <https://github.com/STELLAR-GROUP/hpx/issues/218>4824 <https://github.com/STELLAR-GROUP/hpx/issues/231>4825 <https://github.com/STELLAR-GROUP/hpx/issues/232>4826 <https://github.com/STELLAR-GROUP/hpx/issues/236>4827 <https://github.com/STELLAR-GROUP/hpx/issues/239>4828 <https://github.com/STELLAR-GROUP/hpx/issues/240>4829 <https://github.com/STELLAR-GROUP/hpx/issues/242>4830 <https://github.com/STELLAR-GROUP/hpx/issues/243>4831 <https://github.com/STELLAR-GROUP/hpx/issues/245>4832 <https://github.com/STELLAR-GROUP/hpx/issues/246>

- Issue #252⁴⁸³³ - Problems configuring sheneos with CMake
- Issue #254⁴⁸³⁴ - Lifetime of components doesn't end when client goes out of scope
- Issue #259⁴⁸³⁵ - CMake does not detect that MSVC10 has lambdas
- Issue #260⁴⁸³⁶ - io_service_pool segfault
- Issue #261⁴⁸³⁷ - Late parcel executed outside of pthead
- Issue #263⁴⁸³⁸ - Cannot select allocator with CMake
- Issue #264⁴⁸³⁹ - Fix allocator select
- Issue #267⁴⁸⁴⁰ - Runtime error for hello_world
- Issue #269⁴⁸⁴¹ - pthread_affinity_np test fails to compile
- Issue #270⁴⁸⁴² - Compiler noise due to -Wcast-qual
- Issue #275⁴⁸⁴³ - Problem with configuration tests/include paths on Gentoo
- Issue #325⁴⁸⁴⁴ - Sheneos is 200-400 times slower than the fortran equivalent
- Issue #331⁴⁸⁴⁵ - `hpx::init` and `hpx_main()` should not depend on `program_options`
- Issue #333⁴⁸⁴⁶ - Add doxygen support to CMake for doc toolchain
- Issue #340⁴⁸⁴⁷ - Performance counters for parcels
- Issue #346⁴⁸⁴⁸ - Component loading error when running hello_world in distributed on MSVC2010
- Issue #362⁴⁸⁴⁹ - Missing initializer error
- Issue #363⁴⁸⁵⁰ - Parcel port serialization error
- Issue #366⁴⁸⁵¹ - Parcel buffering leads to types incompatible exception
- Issue #368⁴⁸⁵² - Scalable alternative to rand() needed for *HPX*
- Issue #369⁴⁸⁵³ - IB over IP is substantially slower than just using standard TCP/IP
- Issue #374⁴⁸⁵⁴ - `hpx::lcos::wait` should work with dataflows and arbitrary classes meeting the future interface
- Issue #375⁴⁸⁵⁵ - Conflicting/ambiguous overloads of `hpx::lcos::wait`

⁴⁸³³ <https://github.com/STELLAR-GROUP/hpx/issues/252>

⁴⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/254>

⁴⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/259>

⁴⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/260>

⁴⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/261>

⁴⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/263>

⁴⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/264>

⁴⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/267>

⁴⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/269>

⁴⁸⁴² <https://github.com/STELLAR-GROUP/hpx/issues/270>

⁴⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/275>

⁴⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/325>

⁴⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/331>

⁴⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/333>

⁴⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/340>

⁴⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/346>

⁴⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/362>

⁴⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/363>

⁴⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/366>

⁴⁸⁵² <https://github.com/STELLAR-GROUP/hpx/issues/368>

⁴⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/369>

⁴⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/374>

⁴⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/375>

- Issue #376⁴⁸⁵⁶ - Find_HPX.cmake should set CMake variable HPX_FOUND for out of tree builds
- Issue #377⁴⁸⁵⁷ - ShenEOS interpolate bulk and interpolate_one_bulk are broken
- Issue #379⁴⁸⁵⁸ - Add support for distributed runs under SLURM
- Issue #382⁴⁸⁵⁹ - _Unwind_Word not declared in boost.backtrace
- Issue #387⁴⁸⁶⁰ - Doxygen should look only at list of specified files
- Issue #388⁴⁸⁶¹ - Running make install on an out-of-tree application is broken
- Issue #391⁴⁸⁶² - Out-of-tree application segfaults when running in qsub
- Issue #392⁴⁸⁶³ - Remove HPX_NO_INSTALL option from cmake build system
- Issue #396⁴⁸⁶⁴ - Pragma related warnings when compiling with older gcc versions
- Issue #399⁴⁸⁶⁵ - Out of tree component build problems
- Issue #400⁴⁸⁶⁶ - Out of source builds on Windows: linker should not receive compiler flags
- Issue #401⁴⁸⁶⁷ - Out of source builds on Windows: components need to be linked with hpx_serialization
- Issue #404⁴⁸⁶⁸ - gfortran fails to link automatically when fortran files are present
- Issue #405⁴⁸⁶⁹ - Inability to specify linking order for external libraries
- Issue #406⁴⁸⁷⁰ - Adapt action limits such that dataflow applications work without additional defines
- Issue #415⁴⁸⁷¹ - locality_results is not a member of hpx::components::server
- Issue #425⁴⁸⁷² - Breaking changes to traits::*result wrt std::vector<id_type>
- Issue #426⁴⁸⁷³ - AUTOLOB needs to be updated to support fortran

2.10.22 HPX V0.8.1 (Apr 21, 2012)

This is a point release including important bug fixes for *HPX V0.8.0 (Mar 23, 2012)*.

⁴⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/376>

⁴⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/377>

⁴⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/379>

⁴⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/382>

⁴⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/387>

⁴⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/388>

⁴⁸⁶² <https://github.com/STELLAR-GROUP/hpx/issues/391>

⁴⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/392>

⁴⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/396>

⁴⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/399>

⁴⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/400>

⁴⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/401>

⁴⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/404>

⁴⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/405>

⁴⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/406>

⁴⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/415>

⁴⁸⁷² <https://github.com/STELLAR-GROUP/hpx/issues/425>

⁴⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/426>

General changes

- HPX does not need to be installed anymore to be functional.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this point release:

- Issue #295⁴⁸⁷⁴ - Don't require install path to be known at compile time.
- Issue #371⁴⁸⁷⁵ - Add hpx iostreams to standard build.
- Issue #384⁴⁸⁷⁶ - Fix compilation with GCC 4.7.
- Issue #390⁴⁸⁷⁷ - Remove keep_factory_alive startup call from ShenEOS; add shutdown call to H5close.
- Issue #393⁴⁸⁷⁸ - Thread affinity control is broken.

Bug fixes (commits)

Here is a list of the important commits included in this point release:

- r7642 - External: Fix backtrace memory violation.
- **r7775 - Components: Fix symbol visibility bug with component startup** providers. This prevents one components providers from overriding another components.
- r7778 - Components: Fix startup/shutdown provider shadowing issues.

2.10.23 HPX V0.8.0 (Mar 23, 2012)

We have had roughly 1000 commits since the last release and we have closed approximately 70 tickets (bugs, feature requests, etc.).

General changes

- Improved PBS support, allowing for arbitrary naming schemes of node-hostnames.
- Finished verification of the reference counting framework.
- Implemented decrement merging logic to optimize the distributed reference counting system.
- Restructured the LCO framework. Renamed `hpx::lcos::eager_future<>` and `hpx::lcos::lazy_future<>` into `hpx::lcos::packaged_task` and `hpx::lcos::deferred_packaged_task`. Split `hpx::lcos::promise` into `hpx::lcos::packaged_task` and `hpx::lcos::future`. Added 'local' futures (in namespace `hpx::lcos::local`).
- Improved the general performance of local and remote action invocations. This (under certain circumstances) drastically reduces the number of copies created for each of the parameters and return values.

⁴⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/295>

⁴⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/371>

⁴⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/384>

⁴⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/390>

⁴⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/393>

- Reworked the performance counter framework. Performance counters are now created only when needed, which reduces the overall resource requirements. The new framework allows for much more flexible creation and management of performance counters. The new sine example application demonstrates some of the capabilities of the new infrastructure.
- Added a buildbot-based continuous build system which gives instant, automated feedback on each commit to SVN.
- Added more automated tests to verify proper functioning of *HPX*.
- Started to create documentation for *HPX* and its API.
- Added documentation toolchain to the build system.
- Added dataflow LCO.
- Changed default *HPX* command line options to have `hpx:` prefix. For instance, the former option `--threads` is now `--hpx:threads`. This has been done to make ambiguities with possible application specific command line options as unlikely as possible. See the section *HPX Command Line Options* for a full list of available options.
- Added the possibility to define command line aliases. The former short (one-letter) command line options have been predefined as aliases for backwards compatibility. See the section *HPX Command Line Options* for a detailed description of command line option aliasing.
- Network connections are now cached based on the connected host. The number of simultaneous connections to a particular host is now limited. Parcels are buffered and bundled if all connections are in use.
- Added more refined thread affinity control. This is based on the external library Portable Hardware Locality (HWLOC).
- Improved support for Windows builds with CMake.
- Added support for components to register their own command line options.
- Added the possibility to register custom startup/shutdown functions for any component. These functions are guaranteed to be executed by an *HPX* thread.
- Added two new experimental thread schedulers: `hierarchy_scheduler` and `periodic_priority_scheduler`. These can be activated by using the command line options `--hpx:queuing=hierarchy` or `--hpx:queuing=periodic`.

Example applications

- Graph500 performance benchmark⁴⁸⁷⁹ (thanks to Matthew Anderson for contributing this application).
- GTC (Gyrokinetic Toroidal Code)⁴⁸⁸⁰: a skeleton for particle in cell type codes.
- Random Memory Access: an example demonstrating random memory accesses in a large array
- ShenEOS example⁴⁸⁸¹, 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*.

⁴⁸⁷⁹ <http://www.graph500.org/>

⁴⁸⁸⁰ <http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/gtc/>

⁴⁸⁸¹ <http://stellarcollapse.org/equationofstate>

- Load balancing and work stealing demos.

API changes

- Moved all local LCOs into a separate namespace `hpx::lcos::local` (for instance, `hpx::lcos::local_mutex` is now `hpx::lcos::local::mutex`).
- Replaced `hpx::actions::function` with `hpx::util::function`. Cleaned up related code.
- Removed `hpx::traits::handle_gid` and moved handling of global reference counts into the corresponding serialization code.
- Changed terminology: `prefix` is now called `locality_id`, renamed the corresponding API functions (such as `hpx::get_prefix`, which is now called `hpx::get_locality_id`).
- Adding `hpx::find_remote_localities`, and `hpx::get_num_localities`.
- Changed performance counter naming scheme to make it more bash friendly. The new performance counter naming scheme is now

```
/object{parentname#parentindex/instance#index}/counter#parameters
```

- Added `hpx::get_worker_thread_num` replacing `hpx::threadmanager_base::get_thread_num`.
- Renamed `hpx::get_num_os_threads` to `hpx::get_os_threads_count`.
- Added `hpx::threads::get_thread_count`.
- Restructured the Futures sub-system, renaming types in accordance with the terminology used by the C++11 ISO standard.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #31^{[4882](https://github.com/STELLAR-GROUP/hpx/issues/31)} - Specialize handle_gid<> for examples and tests
- Issue #72^{[4883](https://github.com/STELLAR-GROUP/hpx/issues/72)} - Fix AGAS reference counting
- Issue #104^{[4884](https://github.com/STELLAR-GROUP/hpx/issues/104)} - heartbeat throws an exception when derefing the performance counter it's watching
- Issue #111^{[4885](https://github.com/STELLAR-GROUP/hpx/issues/111)} - throttle causes an exception on the target application
- Issue #142^{[4886](https://github.com/STELLAR-GROUP/hpx/issues/142)} - One failed component loading causes an unrelated component to fail
- Issue #165^{[4887](https://github.com/STELLAR-GROUP/hpx/issues/165)} - Remote exception propagation bug in AGAS reference counting test
- Issue #186^{[4888](https://github.com/STELLAR-GROUP/hpx/issues/186)} - Test credit exhaustion/splitting (e.g. prepare_gid and symbol NS)
- Issue #188^{[4889](https://github.com/STELLAR-GROUP/hpx/issues/188)} - Implement remaining AGAS reference counting test cases
- Issue #258^{[4890](https://github.com/STELLAR-GROUP/hpx/issues/258)} - No type checking of GIDs in stubs classes

⁴⁸⁸² <https://github.com/STELLAR-GROUP/hpx/issues/31>

⁴⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/72>

⁴⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/104>

⁴⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/111>

⁴⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/142>

⁴⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/165>

⁴⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/186>

⁴⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/188>

⁴⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/258>

- Issue #271⁴⁸⁹¹ - Seg fault/shared pointer assertion in distributed code
- Issue #281⁴⁸⁹² - CMake options need descriptive text
- Issue #283⁴⁸⁹³ - AGAS caching broken (gva_cache needs to be rewritten with ICL)
- Issue #285⁴⁸⁹⁴ - HPX_INSTALL root directory not the same as CMAKE_INSTALL_PREFIX
- Issue #286⁴⁸⁹⁵ - New segfault in dataflow applications
- Issue #289⁴⁸⁹⁶ - Exceptions should only be logged if not handled
- Issue #290⁴⁸⁹⁷ - c++11 tests failure
- Issue #293⁴⁸⁹⁸ - Build target for component libraries
- Issue #296⁴⁸⁹⁹ - Compilation error with Boost V1.49rc1
- Issue #298⁴⁹⁰⁰ - Illegal instructions on termination
- Issue #299⁴⁹⁰¹ - gravity aborts with multiple threads
- Issue #301⁴⁹⁰² - Build error with Boost trunk
- Issue #303⁴⁹⁰³ - Logging assertion failure in distributed runs
- Issue #304⁴⁹⁰⁴ - Exception ‘what’ strings are lost when exceptions from decode_parcel are reported
- Issue #306⁴⁹⁰⁵ - Performance counter user interface issues
- Issue #307⁴⁹⁰⁶ - Logging exception in distributed runs
- Issue #308⁴⁹⁰⁷ - Logging deadlocks in distributed
- Issue #309⁴⁹⁰⁸ - Reference counting test failures and exceptions
- Issue #311⁴⁹⁰⁹ - Merge AGAS remote_interface with the runtime_support object
- Issue #314⁴⁹¹⁰ - Object tracking for id_types
- Issue #315⁴⁹¹¹ - Remove handle_gid and handle credit splitting in id_type serialization
- Issue #320⁴⁹¹² - applier::get_locality_id() should return an error value (or throw an exception)
- Issue #321⁴⁹¹³ - Optimization for id_types which are never split should be restored

4891 <https://github.com/STELLAR-GROUP/hpx/issues/271>

4892 <https://github.com/STELLAR-GROUP/hpx/issues/281>

4893 <https://github.com/STELLAR-GROUP/hpx/issues/283>

4894 <https://github.com/STELLAR-GROUP/hpx/issues/285>

4895 <https://github.com/STELLAR-GROUP/hpx/issues/286>

4896 <https://github.com/STELLAR-GROUP/hpx/issues/289>

4897 <https://github.com/STELLAR-GROUP/hpx/issues/290>

4898 <https://github.com/STELLAR-GROUP/hpx/issues/293>

4899 <https://github.com/STELLAR-GROUP/hpx/issues/296>

4900 <https://github.com/STELLAR-GROUP/hpx/issues/298>

4901 <https://github.com/STELLAR-GROUP/hpx/issues/299>

4902 <https://github.com/STELLAR-GROUP/hpx/issues/301>

4903 <https://github.com/STELLAR-GROUP/hpx/issues/303>

4904 <https://github.com/STELLAR-GROUP/hpx/issues/304>

4905 <https://github.com/STELLAR-GROUP/hpx/issues/306>

4906 <https://github.com/STELLAR-GROUP/hpx/issues/307>

4907 <https://github.com/STELLAR-GROUP/hpx/issues/308>

4908 <https://github.com/STELLAR-GROUP/hpx/issues/309>

4909 <https://github.com/STELLAR-GROUP/hpx/issues/311>

4910 <https://github.com/STELLAR-GROUP/hpx/issues/314>

4911 <https://github.com/STELLAR-GROUP/hpx/issues/315>

4912 <https://github.com/STELLAR-GROUP/hpx/issues/320>

4913 <https://github.com/STELLAR-GROUP/hpx/issues/321>

- Issue #322⁴⁹¹⁴ - Command line processing ignored with Boost 1.47.0
- Issue #323⁴⁹¹⁵ - Credit exhaustion causes object to stay alive
- Issue #324⁴⁹¹⁶ - Duplicate exception messages
- Issue #326⁴⁹¹⁷ - Integrate Quickbook with CMake
- Issue #329⁴⁹¹⁸ - --help and --version should still work
- Issue #330⁴⁹¹⁹ - Create pkg-config files
- Issue #337⁴⁹²⁰ - Improve usability of performance counter timestamps
- Issue #338⁴⁹²¹ - Non-std exceptions deriving from std::exceptions in tfunc may be sliced
- Issue #339⁴⁹²² - Decrease the number of send_pending_parcels threads
- Issue #343⁴⁹²³ - Dynamically setting the stack size doesn't work
- Issue #351⁴⁹²⁴ - 'make install' does not update documents
- Issue #353⁴⁹²⁵ - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs
- Issue #355⁴⁹²⁶ - 'make' doesn't do anything after correct configuration
- Issue #356⁴⁹²⁷ - Don't use hpx::util::static_ in topology code
- Issue #359⁴⁹²⁸ - Infinite recursion in hpx::tuple serialization
- Issue #361⁴⁹²⁹ - Add compile time option to disable logging completely
- Issue #364⁴⁹³⁰ - Installation seriously broken in r7443

2.10.24 HPX V0.7.0 (Dec 12, 2011)

We have had roughly 1000 commits since the last release and we have closed approximately 120 tickets (bugs, feature requests, etc.).

⁴⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/322>

⁴⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/323>

⁴⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/324>

⁴⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/326>

⁴⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/329>

⁴⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/330>

⁴⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/337>

⁴⁹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/338>

⁴⁹²² <https://github.com/STELLAR-GROUP/hpx/issues/339>

⁴⁹²³ <https://github.com/STELLAR-GROUP/hpx/issues/343>

⁴⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/351>

⁴⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/353>

⁴⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/355>

⁴⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/356>

⁴⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/359>

⁴⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/361>

⁴⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/364>

General changes

- Completely removed code related to deprecated AGAS V1, started to work on AGAS V2.1.
- Started to clean up and streamline the exposed APIs (see ‘API changes’ below for more details).
- Revamped and unified performance counter framework, added a lot of new performance counter instances for monitoring of a diverse set of internal *HPX* parameters (queue lengths, access statistics, etc.).
- Improved general error handling and logging support.
- Fixed several race conditions, improved overall stability, decreased memory footprint, improved overall performance (major optimizations include native TLS support and ranged-based AGAS caching).
- Added support for running *HPX* applications with PBS.
- Many updates to the build system, added support for gcc 4.5.x and 4.6.x, added C++11 support.
- Many updates to default command line options.
- Added many tests, set up buildbot for continuous integration testing.
- Better shutdown handling of distributed applications.

Example applications

- quickstart/factorial and quickstart/fibonacci, future-recursive parallel algorithms.
- quickstart/hello_world, distributed hello world example.
- quickstart/rma, simple remote memory access example
- quickstart/quicksort, parallel quicksort implementation.
- gtc, gyrokinetic torodial code.
- bfs, breadth-first-search, example code for a graph application.
- sheneos, partitioning of large data sets.
- accumulator, simple component example.
- balancing/os_thread_num, balancing/px_thread_phase, examples demonstrating load balancing and work stealing.

API changes

- Added `hpx::find_all_localities`.
- Added `hpx::terminate` for non-graceful termination of applications.
- Added `hpx::lcos::async` functions for simpler asynchronous programming.
- Added new AGAS interface for handling of symbolic namespace (`hpx::agas::*`).
- Renamed `hpx::components::wait` to `hpx::lcos::wait`.
- Renamed `hpx::lcos::future_value` to `hpx::lcos::promise`.
- Renamed `hpx::lcos::recursive_mutex` to `hpx::lcos::local_recursive_mutex`, `hpx::lcos::mutex` to `hpx::lcos::local_mutex`
- Removed support for Boost versions older than V1.38, recommended Boost version is now V1.47 and newer.
- Removed `hpx::process` (this will be replaced by a real process implementation in the future).

- Removed non-functional LCO code (`hpx::lcos::dataflow`, `hpx::lcos::thunk`, `hpx::lcos::dataflow_variable`).
- Removed deprecated `hpx::naming::full_address`.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #28⁴⁹³¹ - Integrate Windows/Linux CMake code for *HPX* core
- Issue #32⁴⁹³² - `hpx::cout()` should be `hpx::cout`
- Issue #33⁴⁹³³ - AGAS V2 legacy client does not properly handle `error_code`
- Issue #60⁴⁹³⁴ - AGAS: allow for registerid to optionally take ownership of the gid
- Issue #62⁴⁹³⁵ - adaptive1d compilation failure in Fusion
- Issue #64⁴⁹³⁶ - Parcel subsystem doesn't resolve domain names
- Issue #83⁴⁹³⁷ - No error handling if no console is available
- Issue #84⁴⁹³⁸ - No error handling if a hosted locality is treated as the bootstrap server
- Issue #90⁴⁹³⁹ - Add general commandline option `-N`
- Issue #91⁴⁹⁴⁰ - Add possibility to read command line arguments from file
- Issue #92⁴⁹⁴¹ - Always log exceptions/errors to the log file
- Issue #93⁴⁹⁴² - Log the command line/program name
- Issue #95⁴⁹⁴³ - Support for distributed launches
- Issue #97⁴⁹⁴⁴ - Attempt to create a bad component type in AMR examples
- Issue #100⁴⁹⁴⁵ - factorial and factorial_get examples trigger AGAS component type assertions
- Issue #101⁴⁹⁴⁶ - Segfault when `hpx::process::here()` is called in fibonacci2
- Issue #102⁴⁹⁴⁷ - unknown_component_address in int_object_semaphore_client
- Issue #114⁴⁹⁴⁸ - marduk raises assertion with default parameters
- Issue #115⁴⁹⁴⁹ - Logging messages for SMP runs (on the console) shouldn't be buffered

⁴⁹³¹ <https://github.com/STELLAR-GROUP/hpx/issues/28>

⁴⁹³² <https://github.com/STELLAR-GROUP/hpx/issues/32>

⁴⁹³³ <https://github.com/STELLAR-GROUP/hpx/issues/33>

⁴⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/60>

⁴⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/62>

⁴⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/64>

⁴⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/83>

⁴⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/84>

⁴⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/90>

⁴⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/91>

⁴⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/92>

⁴⁹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/93>

⁴⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/95>

⁴⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/97>

⁴⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/100>

⁴⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/101>

⁴⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/102>

⁴⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/114>

⁴⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/115>

- Issue #119⁴⁹⁵⁰ - marduk linking strategy breaks other applications
- Issue #121⁴⁹⁵¹ - pbsdsh problem
- Issue #123⁴⁹⁵² - marduk, dataflow and adaptive1d fail to build
- Issue #124⁴⁹⁵³ - Lower default preprocessing arity
- Issue #125⁴⁹⁵⁴ - Move hpx::detail::diagnostic_information out of the detail namespace
- Issue #126⁴⁹⁵⁵ - Test definitions for AGAS reference counting
- Issue #128⁴⁹⁵⁶ - Add averaging performance counter
- Issue #129⁴⁹⁵⁷ - Error with endian.hpp while building adaptive1d
- Issue #130⁴⁹⁵⁸ - Bad initialization of performance counters
- Issue #131⁴⁹⁵⁹ - Add global startup/shutdown functions to component modules
- Issue #132⁴⁹⁶⁰ - Avoid using auto_ptr
- Issue #133⁴⁹⁶¹ - On Windows hpx.dll doesn't get installed
- Issue #134⁴⁹⁶² - HPX_LIBRARY does not reflect real library name (on Windows)
- Issue #135⁴⁹⁶³ - Add detection of unique_ptr to build system
- Issue #137⁴⁹⁶⁴ - Add command line option allowing to repeatedly evaluate performance counters
- Issue #139⁴⁹⁶⁵ - Logging is broken
- Issue #140⁴⁹⁶⁶ - CMake problem on windows
- Issue #141⁴⁹⁶⁷ - Move all non-component libraries into \$PREFIX/lib/hpx
- Issue #143⁴⁹⁶⁸ - adaptive1d throws an exception with the default command line options
- Issue #146⁴⁹⁶⁹ - Early exception handling is broken
- Issue #147⁴⁹⁷⁰ - Sheneos doesn't link on Linux
- Issue #149⁴⁹⁷¹ - sheneos_test hangs
- Issue #154⁴⁹⁷² - Compilation fails for r5661

⁴⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/119>

⁴⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/121>

⁴⁹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/123>

⁴⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/124>

⁴⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/125>

⁴⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/126>

4956 <https://github.com/STELLAR-GROUP/hpx/issues/128>4957 <https://github.com/STELLAR-GROUP/hpx/issues/129>4958 <https://github.com/STELLAR-GROUP/hpx/issues/130>4959 <https://github.com/STELLAR-GROUP/hpx/issues/131>4960 <https://github.com/STELLAR-GROUP/hpx/issues/132>4961 <https://github.com/STELLAR-GROUP/hpx/issues/133>4962 <https://github.com/STELLAR-GROUP/hpx/issues/134>4963 <https://github.com/STELLAR-GROUP/hpx/issues/135>4964 <https://github.com/STELLAR-GROUP/hpx/issues/137>4965 <https://github.com/STELLAR-GROUP/hpx/issues/139>4966 <https://github.com/STELLAR-GROUP/hpx/issues/140>4967 <https://github.com/STELLAR-GROUP/hpx/issues/141>4968 <https://github.com/STELLAR-GROUP/hpx/issues/143>4969 <https://github.com/STELLAR-GROUP/hpx/issues/146>4970 <https://github.com/STELLAR-GROUP/hpx/issues/147>4971 <https://github.com/STELLAR-GROUP/hpx/issues/149>4972 <https://github.com/STELLAR-GROUP/hpx/issues/154>

- Issue #155⁴⁹⁷³ - Sine performance counters example chokes on chrono headers
- Issue #156⁴⁹⁷⁴ - Add build type to –version
- Issue #157⁴⁹⁷⁵ - Extend AGAS caching to store gid ranges
- Issue #158⁴⁹⁷⁶ - r5691 doesn't compile
- Issue #160⁴⁹⁷⁷ - Re-add AGAS function for resolving a locality to its prefix
- Issue #168⁴⁹⁷⁸ - Managed components should be able to access their own GID
- Issue #169⁴⁹⁷⁹ - Rewrite AGAS future pool
- Issue #179⁴⁹⁸⁰ - Complete switch to request class for AGAS server interface
- Issue #182⁴⁹⁸¹ - Sine performance counter is loaded by other examples
- Issue #185⁴⁹⁸² - Write tests for symbol namespace reference counting
- Issue #191⁴⁹⁸³ - Assignment of read-only variable in point_geometry
- Issue #200⁴⁹⁸⁴ - Seg faults when querying performance counters
- Issue #204⁴⁹⁸⁵ - –ifnames and suffix stripping needs to be more generic
- Issue #205⁴⁹⁸⁶ - –list-* and –print-counter-* options do not work together and produce no warning
- Issue #207⁴⁹⁸⁷ - Implement decrement entry merging
- Issue #208⁴⁹⁸⁸ - Replace the spinlocks in AGAS with hpx::lcos::local_mutexes
- Issue #210⁴⁹⁸⁹ - Add an –ifprefix option
- Issue #214⁴⁹⁹⁰ - Performance test for PX-thread creation
- Issue #216⁴⁹⁹¹ - VS2010 compilation
- Issue #222⁴⁹⁹² - r6045 context_linux_x86.hpp
- Issue #223⁴⁹⁹³ - fibonacci hangs when changing the state of an active thread
- Issue #225⁴⁹⁹⁴ - Active threads end up in the FEB wait queue
- Issue #226⁴⁹⁹⁵ - VS Build Error for Accumulator Client

⁴⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/155>

⁴⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/156>

⁴⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/157>

⁴⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/158>

⁴⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/160>

⁴⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/168>

⁴⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/169>

⁴⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/179>

⁴⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/182>

⁴⁹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/185>

⁴⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/191>

⁴⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/200>

⁴⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/204>

⁴⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/205>

⁴⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/207>

⁴⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/208>

⁴⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/210>

⁴⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/214>

⁴⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/216>

⁴⁹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/222>

⁴⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/223>

⁴⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/225>

⁴⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/226>

- Issue #228⁴⁹⁹⁶ - Move all traits into namespace hpx::traits
- Issue #229⁴⁹⁹⁷ - Invalid initialization of reference in thread_init_data
- Issue #235⁴⁹⁹⁸ - Invalid GID in iostreams
- Issue #238⁴⁹⁹⁹ - Demangle type names for the default implementation of get_action_name
- Issue #241⁵⁰⁰⁰ - C++11 support breaks GCC 4.5
- Issue #247⁵⁰⁰¹ - Reference to temporary with GCC 4.4
- Issue #248⁵⁰⁰² - Seg fault at shutdown with GCC 4.4
- Issue #253⁵⁰⁰³ - Default component action registration kills compiler
- Issue #272⁵⁰⁰⁴ - G++ unrecognized command line option
- Issue #273⁵⁰⁰⁵ - quicksort example doesn't compile
- Issue #277⁵⁰⁰⁶ - Invalid CMake logic for Windows

2.11 Citing HPX

Please cite *HPX* whenever you use it for publications. Use our paper in The Journal of Open Source Software as the main citation for *HPX*:⁵⁰⁰⁷. Use the Zenodo entry for referring to the latest version of *HPX*:⁵⁰⁰⁸. Entries for citing specific versions of *HPX* can also be found at⁵⁰⁰⁹.

2.12 HPX users

A list of institutions and projects using *HPX* can be found on the *HPX* Users⁵⁰¹⁰ page.

2.13 About HPX

2.13.1 History

The development of High Performance ParalleX (*HPX*) began in 2007. At that time, Hartmut Kaiser became interested in the work done by the ParalleX group at the Center for Computation and Technology (CCT)⁵⁰¹¹, a multi-disciplinary research institute at Louisiana State University (LSU)⁵⁰¹². The ParalleX group was working to develop a new and

⁴⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/228>

⁴⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/229>

⁴⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/235>

⁴⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/238>

⁵⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/241>

⁵⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/247>

5002 <https://github.com/STELLAR-GROUP/hpx/issues/248>5003 <https://github.com/STELLAR-GROUP/hpx/issues/253>5004 <https://github.com/STELLAR-GROUP/hpx/issues/272>5005 <https://github.com/STELLAR-GROUP/hpx/issues/273>5006 <https://github.com/STELLAR-GROUP/hpx/issues/277>5007 <https://joss.theoj.org/papers/022e5917b95517dff20cd3742ab95eca>5008 <https://doi.org/10.5281/zenodo.598202>5009 <https://doi.org/10.5281/zenodo.598202>5010 <https://hpx.stellar-group.org/hpx-users/>5011 <https://www.cct.lsu.edu>5012 <https://www.lsu.edu>

experimental execution model for future high performance computing architectures. This model was christened ParalleX. The first implementations of ParalleX were crude, and many of those designs had to be discarded entirely. However, over time the team learned quite a bit about how to design a parallel, distributed runtime system which implements the concepts of ParalleX.

From the very beginning, this endeavour has been a group effort. In addition to a handful of interested researchers, there have always been graduate and undergraduate students participating in the discussions, design, and implementation of *HPX*. In 2011 we decided to formalize our collective research efforts by creating the STE||AR⁵⁰¹³ 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 STE||AR⁵⁰¹⁴ Group was reorganized to become the international community it is today. This consortium of researchers aims to develop stable, sustainable, and scalable tools which will enable application developers to exploit the parallelism latent in the machines of today and tomorrow. Our goal of the *HPX* project is to create a high quality, freely available, open source implementation of ParalleX concepts for conventional and future systems by building a modular and standards conforming runtime system for SMP and distributed application environments. The API exposed by *HPX* is conformant to the interfaces defined by the C++ ISO Standard and adheres to the programming guidelines used by the Boost⁵⁰¹⁵ collection of C++ libraries. We steer the development of *HPX* with real world applications and aim to provide a smooth migration path for domain scientists.

To learn more about STE||AR⁵⁰¹⁶ and ParalleX, see *People* and *Why HPX?*.

2.13.2 People

The STE||AR⁵⁰¹⁷ 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 STE||AR⁵⁰¹⁸ Group is to promote the development of scalable parallel applications by providing a community for ideas, a framework for collaboration, and a platform for communicating these concepts to the broader community.

Our work is focused on building technologies for scalable parallel applications. *HPX*, our general purpose C++ runtime system for parallel and distributed applications, is no exception. We use *HPX* for a broad range of scientific applications, helping scientists and developers to write code which scales better and shows better performance compared to more conventional programming models such as MPI.

HPX is based on *ParalleX* which is a new (and still experimental) parallel execution model aiming to overcome the limitations imposed by the current hardware and the techniques we use to write applications today. Our group focuses on two types of applications - those requiring excellent strong scaling, allowing for a dramatic reduction of execution time for fixed workloads and those needing highest level of sustained performance through massive parallelism. These applications are presently unable (through conventional practices) to effectively exploit a relatively small number of cores in a multi-core system. By extension, these application will not be able to exploit high-end exascale computing systems which are likely to employ hundreds of millions of such cores by the end of this decade.

Critical bottlenecks to the effective use of new generation high performance computing (HPC) systems include:

- *Starvation*: due to lack of usable application parallelism and means of managing it,
- *Overhead*: reduction to permit strong scalability, improve efficiency, and enable dynamic resource management,
- *Latency*: from remote access across system or to local memories,
- *Contention*: due to multicore chip I/O pins, memory banks, and system interconnects.

⁵⁰¹³ <https://stellar-group.org>

⁵⁰¹⁴ <https://stellar-group.org>

⁵⁰¹⁵ <https://www.boost.org/>

⁵⁰¹⁶ <https://stellar-group.org>

⁵⁰¹⁷ <https://stellar-group.org>

⁵⁰¹⁸ <https://stellar-group.org>

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 2.35: Contributors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT) ⁵⁰¹⁹ , Louisiana State University (LSU) ⁵⁰²⁰	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture ⁵⁰²¹ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ⁵⁰²²	thom.heller@gmail.com
Agustin Berge	Center for Computation and Technology (CCT) ⁵⁰²³ , Louisiana State University (LSU) ⁵⁰²⁴	kaballo86@hotmail.com
Mikael Simberg	Swiss National Supercomputing Centre ⁵⁰²⁵	simbergm@cscs.ch
John Biddiscombe	Swiss National Supercomputing Centre ⁵⁰²⁶	biddisco@cscs.ch
Anton Bikineev	Center for Computation and Technology (CCT) ⁵⁰²⁷ , Louisiana State University (LSU) ⁵⁰²⁸	ant.bikineev@gmail.com
Martin Stumpf	Department of Computer Science 3 - Computer Architecture ⁵⁰²⁹ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ⁵⁰³⁰	martin.h.stumpf@gmail.com
Bryce Adelstein Lelbach	NVIDIA ⁵⁰³¹	brycelelbach@gmail.com
Shuangyang Yang	Center for Computation and Technology (CCT) ⁵⁰³² , Louisiana State University (LSU) ⁵⁰³³	syang16@cct.lsu.edu
Jeroen Habraken	Technische Universiteit Eindhoven ⁵⁰³⁴	jhabraken@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT) ⁵⁰³⁵ , Louisiana State University (LSU) ⁵⁰³⁶	sbrandt@cct.lsu.edu
Antoine Tran Tan	Center for Computation and Technology (CCT) ⁵⁰³⁷ , Louisiana State University (LSU) ⁵⁰³⁸	antoine.trantan@lri.fr
Adrian Serio	Center for Computation and Technology (CCT) ⁵⁰³⁹ , Louisiana State University (LSU) ⁵⁰⁴⁰	aserio@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST) ⁵⁰⁴¹ , Indiana University (IU) ⁵⁰⁴²	mbrodowi@indiana.edu

Contributors to this document

Table 2.36: Documentation authors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT) ⁵⁰⁴³ , Louisiana State University (LSU) ⁵⁰⁴⁴	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture ⁵⁰⁴⁵ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ⁵⁰⁴⁶	thom.heller@gmail.com
Bryce Adelstein Lelbach	NVIDIA ⁵⁰⁴⁷	brycelelbach@gmail.com
Vinay C Amatya	Center for Computation and Technology (CCT) ⁵⁰⁴⁸ , Louisiana State University (LSU) ⁵⁰⁴⁹	vamatya@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT) ⁵⁰⁵⁰ , Louisiana State University (LSU) ⁵⁰⁵¹	sbrandt@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST) ⁵⁰⁵² , Indiana University (IU) ⁵⁰⁵³	mbrodowi@indiana.edu
Adrian Serio	Center for Computation and Technology (CCT) ⁵⁰⁵⁴ , Louisiana State University (LSU) ⁵⁰⁵⁵	aserio@cct.lsu.edu

5019 <https://www.cct.lsu.edu>5020 <https://www.lsu.edu>5021 <https://www3.cs.fau.de>5022 <https://www.fau.de>5023 <https://www.cct.lsu.edu>5024 <https://www.lsu.edu>5025 <https://www.csес.ch>5026 <https://www.csес.ch>5027 <https://www.cct.lsu.edu>5028 <https://www.lsu.edu>5029 <https://www3.cs.fau.de>5030 <https://www.fau.de>5031 <https://nvidia.com/>5032 <https://www.cct.lsu.edu>5033 <https://www.lsu.edu>5034 <https://www.tui.nl>5035 <https://www.cct.lsu.edu>5036 <https://www.lsu.edu>5037 <https://www.cct.lsu.edu>5038 <https://www.lsu.edu>5039 <https://www.cct.lsu.edu>5040 <https://www.lsu.edu>5041 <https://pti.iu.edu>5042 <https://www.iu.edu>5043 <https://www.cct.lsu.edu>5044 <https://www.lsu.edu>5045 <https://www3.cs.fau.de>5046 <https://www.fau.de>5047 <https://nvidia.com/>5048 <https://www.cct.lsu.edu>5049 <https://www.lsu.edu>5050 <https://www.cct.lsu.edu>5051 <https://www.lsu.edu>5052 <https://pti.iu.edu>5053 <https://www.iu.edu>5054 <https://www.cct.lsu.edu>5055 <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.

- Srinivas Yadav, for his work on SIMD support in algorithms before and during Google Summer of Code 2021.
- Akhil Nair, for his work on adapting algorithms to C++20 before and during Google Summer of Code 2021.
- Alexander Toktarev, for updating the parallel algorithm customization points to use `tag_fallback_invoke` for the default implementations.
- Brice Goglin, for reporting and helping fix issues related to the integration of hwloc in *HPX*.
- Giannis Gonidelis, for his work on the ranges adaptation during the Google Summer of Code 2020.
- Auriane Reverdell (Swiss National Supercomputing Centre⁵⁰⁵⁶), for her tireless work on refactoring our CMake setup and modularizing *HPX*.
- Christopher Hinz, for his work on refactoring our CMake setup.
- Weile Wei, for fixing *HPX* builds with CUDA on Summit.
- Severin Strobl, for fixing our CMake setup related to linking and adding new entry points to the *HPX* runtime.
- Rebecca Stobaugh, for her major documentation review and contributions during and after the 2019 Google Season of Documentation.
- Jan Melech, for adding automatic serialization of simple structs.
- Austin McCartney, for adding concept emulation of the Ranges TS bidirectional and random access iterator concepts.
- Marco Diers, reporting and fixing issues related PMIx.
- Maximilian Bremer, for reporting multiple issues and extending the component migration tests.
- Piotr Mikolajczyk, for his improvements and fixes to the set and count algorithms.
- Grant Rostig, for reporting several deficiencies on our web pages.
- Jakub Golinowski, for implementing an *HPX* backend for OpenCV and in the process improving documentation and reporting issues.
- Mikael Simberg (Swiss National Supercomputing Centre⁵⁰⁵⁷), 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.

⁵⁰⁵⁶ <https://www.cscs.ch>

⁵⁰⁵⁷ <https://www.cscs.ch>

- Denis Blank, who re-wrote our unwrapped function to accept plain values arbitrary containers, and properly deal with nested futures.
- Ajai V. George, who implemented several of the parallel algorithms.
- Taeguk Kwon, who worked on implementing parallel algorithms as well as adapting the parallel algorithms to the Ranges TS.
- Zach Byerly ([Louisiana State University \(LSU\)](#)⁵⁰⁵⁸), 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 \(LSU\)](#)⁵⁰⁵⁹), 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 \(LSU\)](#)⁵⁰⁶⁰), who contributed the prefetching iterators and the persistent auto chunking executor parameters implementation.
- Marcin Copik, who worked on implementing GPU support for C++AMP and HCC. He also worked on implementing a HCC backend for *HPX.Compute*.
- Minh-Khanh Do, who contributed the implementation of several segmented algorithms.
- Bibek Wagle ([Louisiana State University \(LSU\)](#)⁵⁰⁶¹), 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](#)⁵⁰⁶²) with *HPX*. He reported various problems and submitted several patches to fix issues allowing for a better integration with [LibGeoDecomp](#)⁵⁰⁶³.
- 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 \(LSU\)](#)⁵⁰⁶⁴), 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](#)⁵⁰⁶⁵ C/C++ package manager.
- Vinay C Amatya ([Louisiana State University \(LSU\)](#)⁵⁰⁶⁶), who contributed to the documentation and provided some of the *HPX* examples.

⁵⁰⁵⁸ <https://www.lsu.edu>⁵⁰⁵⁹ <https://www.lsu.edu>⁵⁰⁶⁰ <https://www.lsu.edu>⁵⁰⁶¹ <https://www.lsu.edu>⁵⁰⁶² <https://www.libgeodecomp.org/>⁵⁰⁶³ <https://www.libgeodecomp.org/>⁵⁰⁶⁴ <https://www.lsu.edu>⁵⁰⁶⁵ <https://www.conan.io/>⁵⁰⁶⁶ <https://www.lsu.edu>

- Kevin Huck and Nick Chaimov ([University of Oregon](https://uoregon.edu/)⁵⁰⁶⁷), who contributed the integration of APEX (Autonomic Performance Environment for eXascale) with *HPX*.
- Francisco Jose Tapia, who helped with implementing the parallel sort algorithm for *HPX*.
- Patrick Diehl, who worked on implementing CUDA support for our companion library targeting GPGPUs ([HPXCL](https://github.com/STELLAR-GROUP/hpxcl/)⁵⁰⁶⁸).
- 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/)⁵⁰⁶⁹) 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)⁵⁰⁷⁰).
- Anuj Sharma and Christopher Bross ([Department of Computer Science 3 - Computer Architecture](https://www.cs.fau.de/)⁵⁰⁷¹), who worked on *HPX* in the context of the [Google Summer of Code](https://www.nvidia.com/object/cuda_home_new.html)⁵⁰⁷² program 2014.
- Martin Stumpf ([Department of Computer Science 3 - Computer Architecture](https://www.cs.fau.de/)⁵⁰⁷³), who rebuilt our contiguous testing infrastructure (see the [HPX Buildbot Website](https://github.com/STELLAR-GROUP/hpxcl/)⁵⁰⁷⁴). Martin is also working on [HPXCL](https://github.com/STELLAR-GROUP/hpxcl/)⁵⁰⁷⁵ (mainly all work related to [OpenCL](https://www.khronos.org/opencl/)⁵⁰⁷⁶) and implementing an *HPX* backend for [POCL](https://www.khronos.org/opencl/)⁵⁰⁷⁷, a portable computing language solution based on OpenCL⁵⁰⁷⁸.
- Grant Mercer ([University of Nevada, Las Vegas](https://uoregon.edu/)⁵⁰⁷⁹), who helped creating many of the parallel algorithms (as proposed by [N4313](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html)⁵⁰⁸⁰).
- Damond Howard ([Louisiana State University \(LSU\)](https://www.cs.fau.de/)⁵⁰⁸¹), who works on [HPXCL](https://github.com/STELLAR-GROUP/hpxcl/)⁵⁰⁸² (mainly all work related to [CUDA](https://www.khronos.org/opencl/)⁵⁰⁸³).
- Christoph Junghans (Los Alamos National Lab), who helped making our buildsystem more portable.
- Antoine Tran Tan ([Laboratoire de Recherche en Informatique, Paris](https://www.cs.fau.de/)), who worked on integrating *HPX* as a backend for [NT2](https://www.cs.fau.de/)⁵⁰⁸⁴. 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.cs.fau.de/)⁵⁰⁸⁵), 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.

⁵⁰⁶⁷ <https://uoregon.edu/>

⁵⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpxcl/>

⁵⁰⁶⁹ <https://www.cct.lsu.edu/>

⁵⁰⁷⁰ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

⁵⁰⁷¹ <https://www3.cs.fau.de>

⁵⁰⁷² <https://developers.google.com/open-source/soc/>

⁵⁰⁷³ <https://www3.cs.fau.de>

⁵⁰⁷⁴ <http://rostam.cct.lsu.edu/>

⁵⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpxcl/>

⁵⁰⁷⁶ <https://www.khronos.org/opencl/>

⁵⁰⁷⁷ <https://portablecl.org/>

⁵⁰⁷⁸ <https://www.khronos.org/opencl/>

⁵⁰⁷⁹ <https://www.unlv.edu>

⁵⁰⁸⁰ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html>

⁵⁰⁸¹ <https://www.lsu.edu>

⁵⁰⁸² <https://github.com/STELLAR-GROUP/hpxcl/>

⁵⁰⁸³ https://www.nvidia.com/object/cuda_home_new.html

⁵⁰⁸⁴ <https://www.numscale.com/nt2/>

⁵⁰⁸⁵ <https://www.cscs.ch>

- 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)⁵⁰⁸⁶), who contributed the description of the different *HPX* thread scheduler policies and is working on the performance analysis of our thread scheduling subsystem.
- Lars Viklund, whose wit, passion for testing, and love of odd architectures has been an amazing contribution to our team. He has also contributed platform specific patches for FreeBSD and MSVC12.
- Agustin Berge, who contributed patches fixing some very nasty hidden template meta-programming issues. He rewrote large parts of the API elements ensuring strict conformance with the C++ ISO Standard.
- Anton Bikineev for contributing changes to make using `boost::lexical_cast` safer, he also contributed a thread safety fix to the iostreams module. He also contributed a complete rewrite of the serialization infrastructure replacing Boost.Serialization inside *HPX*.
- Pyry Jakkola, who contributed the Mac OS build system and build documentation on how to build *HPX* using Clang and `libc++`.
- Mario Mulansky, who created an *HPX* backend for his Boost.Odeint library, and who submitted several test cases allowing us to reproduce and fix problems in *HPX*.
- Rekha Raj, who contributed changes to the description of the Windows build instructions.
- Jeremy Kemp how worked on an *HPX* OpenMP backend and added regression tests.
- Alex Nagelberg for his work on implementing a C wrapper API for *HPX*.
- Chen Guo, helvihartmann, Nicholas Pezolano, and John West who added and improved examples in *HPX*.
- Joseph Kleinhenz, Markus Elfring, Kirill Kropivnyansky, Alexander Neundorf, Bryant Lam, and Alex Hirsch who improved our CMake.
- Tapasweni Pathak, Praveen Velliengiri, Jean-Loup Tastet, Michael Levine, Aalekh Nigam, HadrienG2, Prayag Verma, Islada, Alex Myczko, and Avyav Kumar who improved the documentation.
- Jayesh Badwaik, J. F. Bastien, Christoph Garth, Christopher Hinz, Brandon Kohn, Mario Lang, Maikel Nadolski, pierrele, hendrx, Dekken, woodmeister123, xaguilar, Andrew Kemp, Dylan Stark, Matthew Anderson, Jeremy Wilke, Jiazheng Yuan, CyberDrudge, david8dixon, Maxwell Reeser, Raffaele Solca, Marco Ippolito, Jules Penuchot, Weile Wei, Severin Strobl, Kor de Jong, albestro, Jeff Trull, Yuri Victorovich, and Gregor Daiß who contributed to the general improvement of *HPX*.

HPX Funding Acknowledgements⁵⁰⁸⁷ lists current and past funding sources for *HPX*.

⁵⁰⁸⁶ <https://www.nmsu.edu>

⁵⁰⁸⁷ <https://hpx.stellar-group.org/funding-acknowledgements/>

**CHAPTER
THREE**

INDEX

- genindex