
HPX Documentation

master

The STE||AR Group

November 19, 2022

USER DOCUMENTATION

1	What is <i>HPX</i>?	3
2	What's so special about <i>HPX</i>?	5
2.1	Quick start	5
2.2	Examples	10
2.3	Manual	40
2.4	Terminology	223
2.5	Why <i>HPX</i> ?	224
2.6	Additional material	229
2.7	Overview	230
2.8	API reference	256
2.9	Contributing to <i>HPX</i>	1413
2.10	Releases	1422
2.11	Citing <i>HPX</i>	1683
2.12	<i>HPX</i> users	1683
2.13	About <i>HPX</i>	1683
3	Index	1693

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.

You can find a comprehensive list of contact options on [Support for deploying and using HPX](#)⁵.

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

There are also available a [PDF](#) version of this documentation as well as a [Single HTML Page](#).

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>

⁵ <https://github.com/STELLAR-GROUP/hpx/blob/master/.github/SUPPORT.md>

**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`⁶ dependency manager:

```
$ vcpkg install hpx
```

2. **Spack**

Another way to install *HPX* is using `Spack`⁷:

```
$ spack install hpx
```

⁶ <https://github.com/Microsoft/vcpkg>

⁷ <https://spack.readthedocs.io/en/latest/>

3. Fedora

Installation can be done with Fedora⁸ as well:

```
$ dnf install hpx*
```

4. Arch Linux

HPX is available in the [Arch User Repository \(AUR\)](#)⁹ as `hpx` too.

More information or alternatives regarding the installation can be found in the [Building HPX](#), 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" << std::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.

⁸ <https://fedoraproject.org/wiki/DNF>

⁹ https://wiki.archlinux.org/title/Arch_User_Repository

- `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](#).

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>

#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.
```

(continues on next page)

(continued from previous page)

```

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;

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

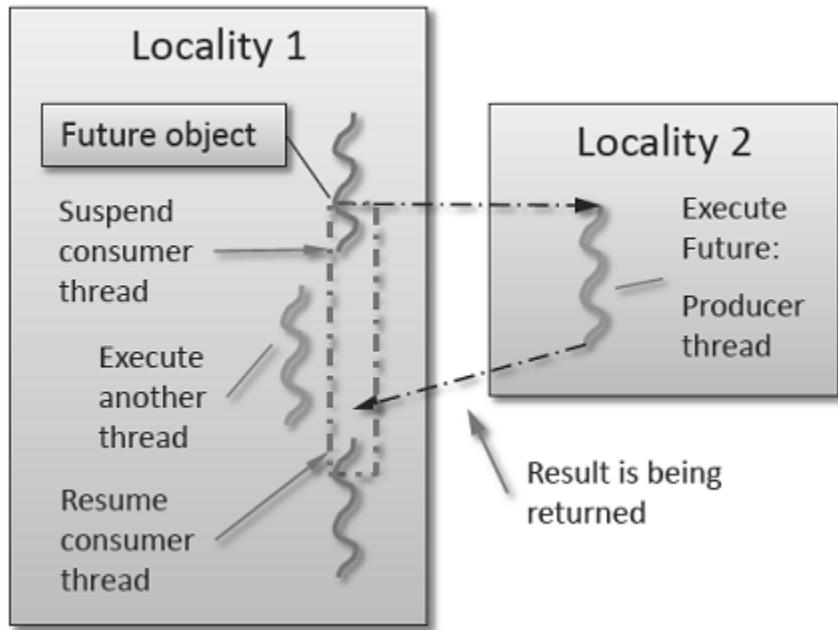


Fig. 2.1: Schematic of a future execution.

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 [Building HPX](#) 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](#)¹⁰ 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](#)¹¹ for more details). `hpx::init` calls `hpx_main()` after setting up *HPX*, which is where the logic of our program is encoded.

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

(continues on next page)

¹⁰ https://www.boost.org/doc/html/program_options.html

¹¹ <https://www.boost.org/doc/>

(continued from previous page)

```

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

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

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

```

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 [Building HPX](#) 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<int>()>default_value(0),
         "Lower limit of range of values")
        ("u",
         hpx::program_options::value<int>()>default_value(10),
         "Upper limit of range of values");
    // clang-format on
    hpx::local::init_params init_args;
    init_args.desc_cmdline = cmdline;
```

(continues on next page)

(continued from previous page)

```

    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);

    // 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
    int lower = vm["l"].as<int>();
    int upper = vm["u"].as<int>();

    // 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::experimental::for_loop(hpx::execution::par, 0, rowsA, [&](auto i) {
        hpx::experimental::for_loop(0, colsB, [&](auto j) {
            R[i * colsR + j] = 0;
            hpx::experimental::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");

```

(continues on next page)

(continued from previous page)

```

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`¹². 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::experimental::for_loop` combined with a parallel execution policy `hpx::execution::par` as the outer loop of the multiplication. Note that the execution of `hpx::experimental::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)
{
    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 [Building HPX](#) for information on configuring and building *HPX*) and enter:

```
$ make examples.quickstart.fibonacci
```

To run the program type:

¹² https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

```
$ ./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;
```

(continues on next page)

(continued from previous page)

```
// 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::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);
```

(continues on next page)

(continued from previous page)

```

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 [Building HPX](#) 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::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());

    for (hpx::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_action` is declared here:

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

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

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

```
void hello_world_foreman()
{
```

(continues on next page)

(continued from previous page)

```

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

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

// As long as there are still elements in the set, we must keep scheduling
// HPX-threads. Because HPX features work-stealing task schedulers, we have
// no way of enforcing which worker OS-thread will actually execute
// each HPX-thread.
while (!attendance.empty())
{
    // Each iteration, we create a task for each element in the set of
// OS-threads that have not said "Hello world". Each of these tasks
// is encapsulated in a future.
std::vector<hpx::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 that we can query to determine if the task has completed.
        //
        // We give the task a hint to run on a particular worker thread
// (core) and suggest binding the scheduled thread to the given
// core, but no guarantees are given by the scheduler that the task
// will actually run on that worker thread. It will however try as
// hard as possible to place the new task on the given worker
// thread.
        hpx::execution::parallel_executor exec(
            hpx::threads::thread_priority::bound);

        hpx::threads::thread_schedule_hint hint(
            hpx::threads::thread_schedule_hint_mode::thread,
            static_cast<std::int16_t>(worker));

        futures.push_back(
            hpx::async(hpx::execution::experimental::with_hint(exec, hint),
                      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::spinlock mtx;
}

```

(continues on next page)

(continued from previous page)

```

    hpx::wait_each(hpx::unwrapping([&](std::size_t t) {
        if (std::size_t(-1) != t)
        {
            std::lock_guard<hpx::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";

        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 *Building HPX* 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 [Building HPX](#) 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),
```

(continues on next page)

(continued from previous page)

```

"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::id_type here = hpx::find_here();

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

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

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

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

    for (int i = 0; i < t; ++i)
    {
        shared_future<double> interest =

```

(continues on next page)

(continued from previous page)

```

        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. 2.2 below.

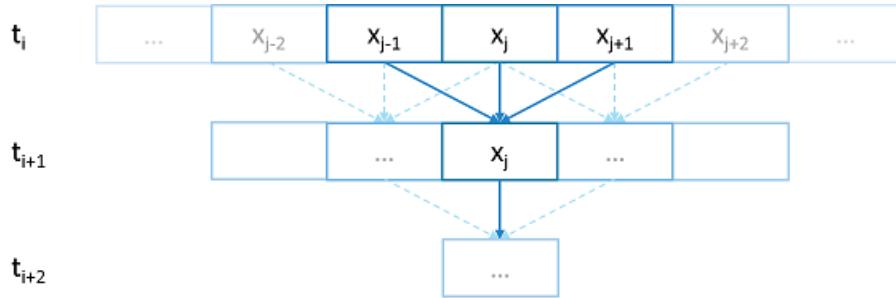


Fig. 2.2: Heat diffusion example program flow.

We parallelize this code over the following eight examples:

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

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

(continues on next page)

(continued from previous page)

```

// do all the work on 'nx' data points for 'nt' time steps
space do_work(std::size_t nx, std::size_t nt)
{
    // U[t][i] is the state of position i at time t.
    std::vector<space> U(2);
    for (space& s : U)
        s.resize(nx);

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

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

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

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

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

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

```

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

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

```

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)
    {

```

(continues on next page)

(continued from previous page)

```

    return middle + (k * dt / (dx * dx)) * (left - 2 * middle + right);
}

// do all the work on 'nx' data points for 'nt' time steps
hpx::future<space> do_work(std::size_t nx, std::size_t nt)
{
    using hpx::dataflow;
    using hpx::unwrapping;

    // 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, nl)], nx, double(i));
```

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

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

```
// 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();
        }));
}
```

(continues on next page)

(continued from previous page)

```

next[0] = heat(l[size - 1], m[0], m[1]);
next[size - 1] = heat(m[size - 2], m[size - 1], r[0]);

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

```

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

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

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

2.2.8 Serializing user-defined types

In order to facilitate the sending and receiving of complex datatypes *HPX* provides a serialization abstraction.

Just like boost, hpx allows users to serialize user-defined types by either providing the serializer as a member function or defining the serialization as a free function.

Unlike Boost HPX doesn't acknowledge second unsigned int parameter, it is solely there to preserve API compatibility with Boost Serialization

This tutorial was heavily inspired by Boost's serialization concepts¹³.

¹³ https://www.boost.org/doc/libs/1_79_0/libs/serialization/doc/serialization.html

Setup

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

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

```
$ make examples.quickstart.custom_serialization
```

To run the program type:

```
$ ./bin/custom_serialization
```

This should print:

```
Rectangle(Point(x=0,y=0),Point(x=0,y=5))
gravity.g = 9.81%
```

Serialization Requirements

In order to serialize objects in *HPX*, at least one of the following criteria must be met:

In the case of default constructible objects:

- The object is an empty type.
- Has a serialization function as shown in this tutorial.
- All members are accessible publicly and they can be used in structured binding contexts.

Otherwise:

- They need to have special serialization support.

Member function serialization

```
struct point_member_serialization
{
    int x{0};
    int y{0};

    // Required when defining the serialization function as private
    // In this case it isn't
    // Provides serialization access to HPX
    friend class hpx::serialization::access;

    // Second argument exists solely for compatibility with boost serialize
    // it is NOT processed by HPX in any way.
    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        // clang-format off
        ar & x & y;
        // clang-format on
    }
}
```

(continues on next page)

(continued from previous page)

```
};

// Allow bitwise serialization
HPX_IS_BITWISE_SERIALIZABLE(point_member_serialization)
```

Notice that `point_member_serialization` is defined as bitwise serializable (see [Bitwise serialization for bitwise copyable data](#) for more details). HPX is also able to recursively serialize composite classes and structs given that its members are serializable.

```
struct rectangle_member_serialization
{
    point_member_serialization top_left;
    point_member_serialization lower_right;

    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        // clang-format off
        ar & top_left & lower_right;
        // clang-format on
    }
};
```

Free function serialization

In order to decouple your models from HPX, HPX also allows for the definition of free function serializers.

```
struct rectangle_free
{
    point_member_serialization top_left;
    point_member_serialization lower_right;
};

template <typename Archive>
void serialize(Archive& ar, rectangle_free& pt, const unsigned int)
{
    // clang-format off
    ar & pt.lower_right & pt.top_left;
    // clang-format on
}
```

Even if you can't modify a class to befriend it, you can still be able to serialize your class provided that your class is default constructable and you are able to reconstruct it yourself.

```
class point_class
{
public:
    point_class(int x, int y)
        : x(x)
        , y(y)
    {
```

(continues on next page)

(continued from previous page)

```

}

point_class() = default;

[[nodiscard]] int get_x() const noexcept
{
    return x;
}

[[nodiscard]] int get_y() const noexcept
{
    return y;
}

private:
    int x;
    int y;
};

template <typename Archive>
void load(Archive& ar, point_class& pt, const unsigned int)
{
    int x, y;
    ar >> x >> y;
    pt = point_class(x, y);
}

template <typename Archive>
void save(Archive& ar, point_class const& pt, const unsigned int)
{
    ar << pt.get_x() << pt.get_y();
}

// This tells HPX that you have split your serialize function into
// load and save
HPX_SERIALIZATION_SPLIT_FREE(point_class)

```

Serializing non default constructable classes

Some classes don't provide any default constructor.

```

class planet_weight_calculator
{
public:
    explicit planet_weight_calculator(double g)
        : g(g)
    {

        template <class Archive>
        friend void save_construct_data(

```

(continues on next page)

(continued from previous page)

```

Archive&, planet_weight_calculator const*, unsigned int);

[[nodiscard]] double get_g() const
{
    return g;
}

private:
    // Provides serialization access to HPX
    friend class hpx::serialization::access;
    template <class Archive>
    void serialize(Archive&, const unsigned int)
    {
        // Serialization will be done in the save_construct_data
        // Still needs to be defined
    }

    double g;
};

```

In this case you have to define a `save_construct_data` and `load_construct_data` in which you do the serialization yourself.

```

template <class Archive>
inline void save_construct_data(Archive& ar,
    planet_weight_calculator const* weight_calc, const unsigned int)
{
    ar << weight_calc->g;      // Do all of your serialization here
}

template <class Archive>
inline void load_construct_data(
    Archive& ar, planet_weight_calculator* weight_calc, const unsigned int)
{
    double g;
    ar >> g;

    // ::new(ptr) construct new object at given address
    ::new (weight_calc) planet_weight_calculator(g);
}

```

Bitwise serialization for bitwise copyable data

When sending non arithmetic types not defined by `std::is_arithmetic`¹⁴, HPX has to (de)serialize each object separately. However, if the class you are trying to send classes consists only of bitwise copyable datatypes, you may mark your class as such. Then HPX will serialize your object bitwise instead of element wise. This has enormous benefits, especially when sending a vector/array of your class. To define your class as such you need to call `HPX_IS_BITWISE_SERIALIZABLE(T)` with your desired custom class.

¹⁴ https://en.cppreference.com/w/cpp/types/is_arithmetic

```
struct point_member_serialization
{
    int x{0};
    int y{0};

    // Required when defining the serialization function as private
    // In this case it isn't
    // Provides serialization access to HPX
    friend class hpx::serialization::access;

    // Second argument exists solely for compatibility with boost serialize
    // it is NOT processed by HPX in any way.
    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        // clang-format off
        ar & x & y;
        // clang-format on
    }
};

// Allow bitwise serialization
HPX_IS_BITWISE_SERIALIZABLE(point_member_serialization)
```

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 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++) ¹⁵	8.0
clang: a C language family frontend for LLVM ¹⁶	10.0
Visual C++ ¹⁷ (x64)	2019

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 ¹⁸	3.18
Required Libraries	Boost ¹⁹	1.71.0
	Portable Hardware Locality (HWLOC) ²⁰	1.5
	Asio ²¹	1.12.0

The most important dependencies are Boost²² and Portable Hardware Locality (HWLOC)²³. The installation of Boost is described in detail in Boost's Getting Started²⁴ document. A recent version of hwloc is required in order to support thread pinning and NUMA awareness and can be found in Hwloc Downloads²⁵.

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.

¹⁵ <https://gcc.gnu.org>

¹⁶ <https://clang.llvm.org/>

¹⁷ <https://msdn.microsoft.com/en-us/visualc/default.aspx>

¹⁸ <https://www.cmake.org>

¹⁹ <https://www.boost.org/>

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

²¹ <https://think-async.com/asio/>

²² <https://www.boost.org/>

²³ <https://www.open-mpi.org/projects/hwloc/>

²⁴ https://www.boost.org/more/getting_started/index.html

²⁵ <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> ²⁶	1.7.1
<code>jemalloc</code> ²⁷	2.1.0
<code>mi-malloc</code> ²⁸	1.0.0
Performance Application Programming Interface (PAPI)	

2.3.2 Getting *HPX*

Download a tarball of the latest release from [HPX Downloads](#)²⁹ 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 1.9.0
```

2.3.3 Building *HPX*

Basic information

The build system for *HPX* is based on [CMake](#)³⁰, 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](#)³¹.

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.

²⁶ <https://code.google.com/p/gperftools>

²⁷ <http://jemalloc.net>

²⁸ <http://microsoft.github.io/mimalloc/>

²⁹ <https://hpx.stellar-group.org/downloads/>

³⁰ <https://www.cmake.org>

³¹ <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](#)³² 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.

³² <https://uo-oaciss.github.io/apex/quickstarthpx/>

HPX_WITH_TESTS

Build tests.

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

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](#)³³). 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](#)³⁴, 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 ..
```

³³ https://cmake.org/cmake/help/latest/variable/CMAKE_BUILD_TYPE.html

³⁴ <https://www.cmake.org>

- 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:

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

For instance:

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

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

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

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

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

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

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

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

Windows

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

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

- Download the CMake V3.18.1 installer (or latest version) from [here](#)³⁵
- Download the hwloc V1.11.0 (or the latest version) from [here](#)³⁶ and unpack it.
- Download the latest Boost libraries from [here](#)³⁷ and unpack them.

³⁵ <https://blog.kitware.com/cmake-3-18-1-available-for-download/>

³⁶ <http://www.open-mpi.org/software/hwloc/v1.11/downloads/hwloc-win64-build-1.11.0.zip>

³⁷ <https://www.boost.org/users/download/>

- Build the Boost DLLs and LIBs by using these commands from Command Line (or PowerShell). Open CMD/PowerShell inside the Boost dir and type in:

```
.\bootstrap.bat
```

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

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

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

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

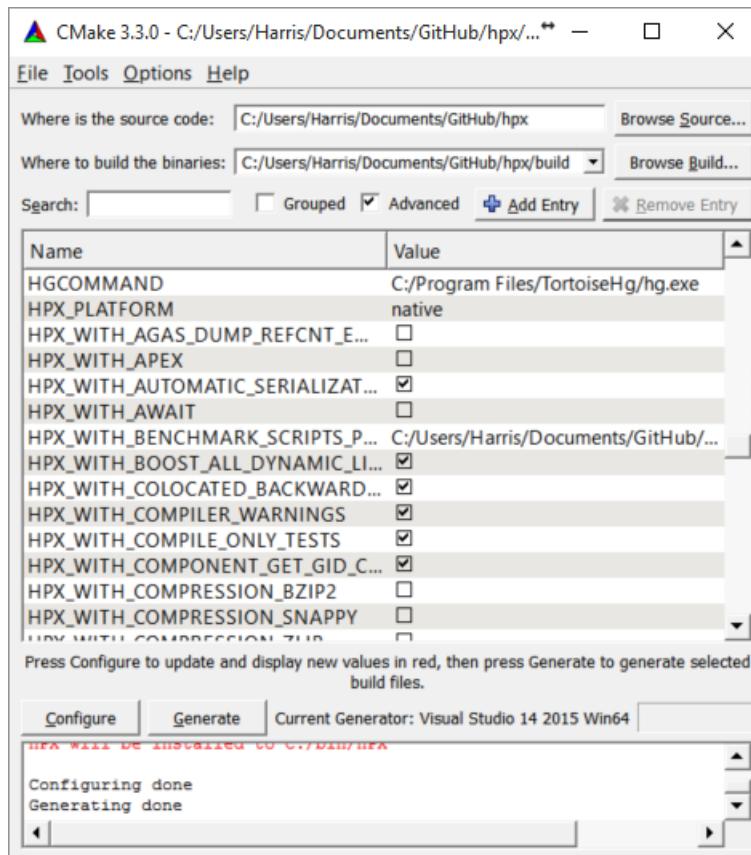


Fig. 2.3: Example CMake path settings.

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, ASIO_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.
- ASIO_ROOT the *HPX* root directory of the unpacked ASIO files. Alternatively use HPX_WITH_FETCH_ASIO with value True.
- 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:

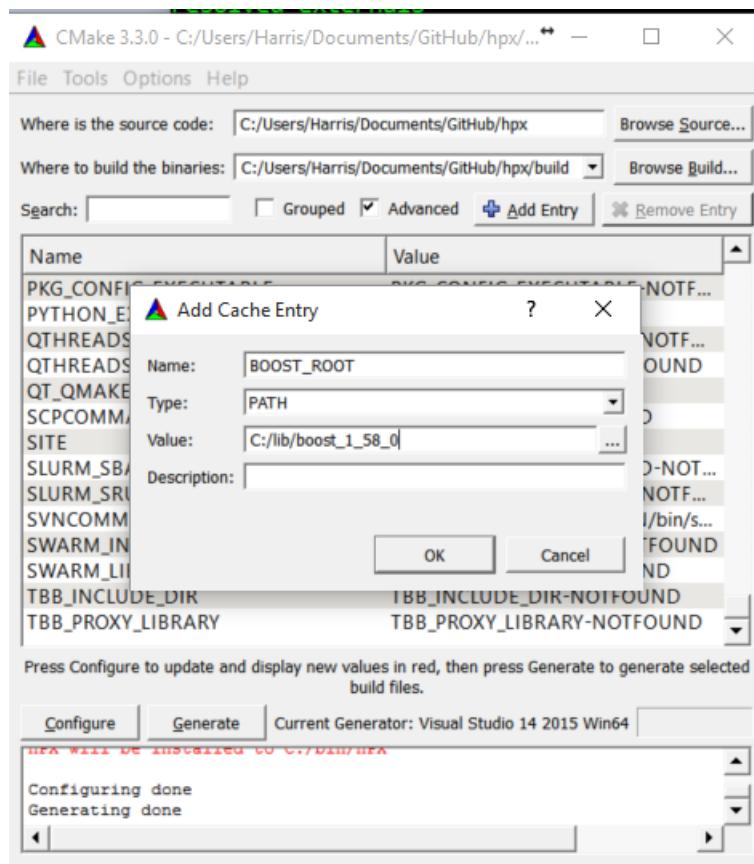


Fig. 2.4: Example CMake adding entry.

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 roottdir. 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:

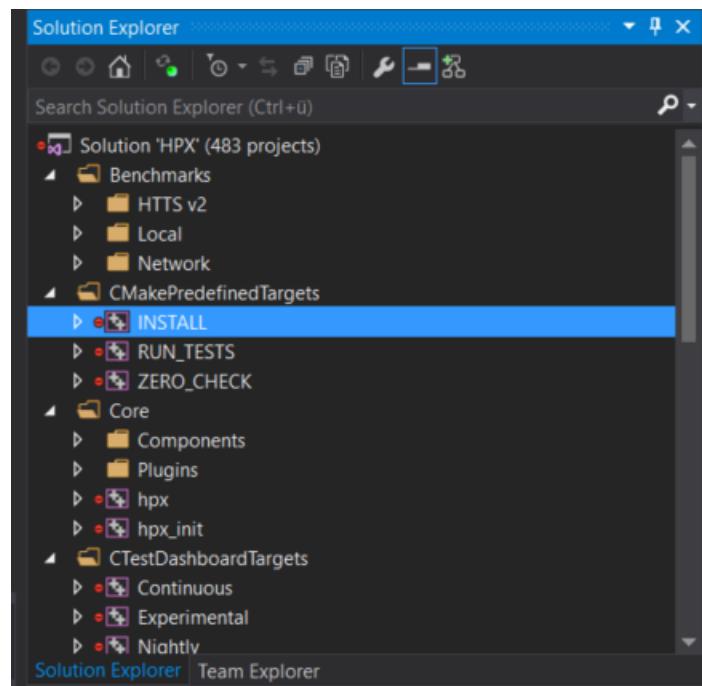


Fig. 2.5: Visual Studio INSTALL target.

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
```

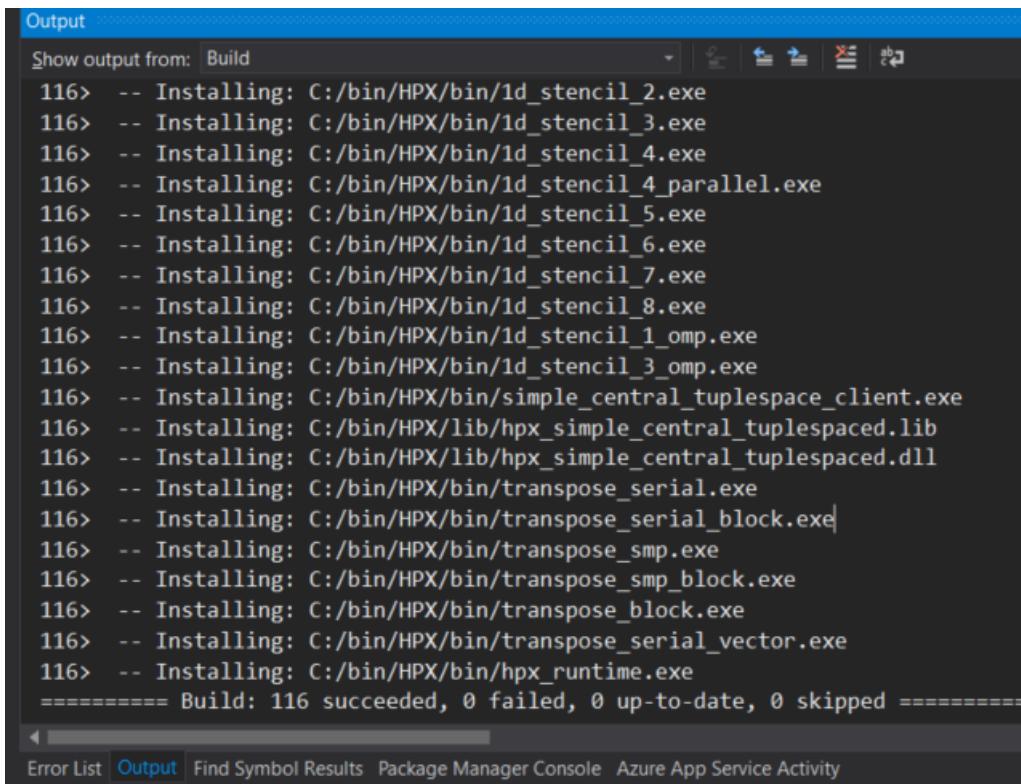
To control which tests to run use `ctest`:

- 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:

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



The screenshot shows the Visual Studio Output window with the title 'Output' at the top. Below it, a dropdown menu says 'Show output from: Build'. The main area contains a log of 116 installations of various HPX executables and libraries. The log includes names like '1d_stencil_2.exe', '1d_stencil_3.exe', '1d_stencil_4.exe', etc., followed by 'simple_central_tuplespace_client.exe', 'hpx_simple_central_tuplespaced.lib', and 'hpx_simple_central_tuplespaced.dll'. It also lists transpose-related executables such as 'transpose_serial.exe', 'transpose_serial_block.exe', 'transpose_smp.exe', 'transpose_smp_block.exe', and 'transpose_block.exe'. The log concludes with a summary: 'Build: 116 succeeded, 0 failed, 0 up-to-date, 0 skipped ======='.

Fig. 2.6: Visual Studio build output.

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

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

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

2.3.4 CMake variables used to configure HPX

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

Variables that influence how *HPX* is built

The options are split into these categories:

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

Generic options

- `HPX_WITH_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_BACKEND:STRING*
- *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 using Vc library (default: ON)

HPX_WITH_DATAPAR_BACKEND:STRING

Define which vectorization library should be used. Options are: VC, EVE, STD_EXPERIMENTAL SIMD, SVE; NONE

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: ON)

HPX_WITH_NICE_THREADLEVEL:BOOL

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

HPX_WITH_PARCEL_COALESCING:BOOL

Enable the parcel coalescing plugin (default: ON).

HPX_WITH_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_FETCH_LCI:BOOL*
- *HPX_WITH_IO_COUNTERS:BOOL*
- *HPX_WITH_LCI_TAG:STRING*
- *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_FETCH_LCI:BOOL

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

HPX_WITH_IO_COUNTERS:BOOL

Enable IO counters (default: ON)

HPX_WITH_LCI_TAG:STRING

LCI repository tag or branch

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_COUNTERS:BOOL*
- *HPX_WITH_PARCELPORT_LCI: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_COUNTERS:BOOL

Enable performance counters reporting parcelport statistics.

HPX_WITH_PARCELPORT_LCI:BOOL

Enable the LCI based parcelport.

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_DATASTRUCTURES_WITH_ADAPT_STD_VARIANT: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_SERIALIZATION_WITH_SUPPORTS_ENDIANESS:BOOL*
- *HPX_TOPOLOGY_WITH_ADDITIONAL_HWLOC_TESTING:BOOL*

- `HPX_WITH_POWER_COUNTER:BOOL`

HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL

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

HPX_DATASTRUCTURES_WITH_ADAPT_STD_VARIANT:BOOL

Enable compatibility of hpx::get with std::variant.

(default: OFF)

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: OFF)

HPX_SERIALIZATION_WITH_SUPPORTS_ENDIANESS:BOOL

Support endian conversion on inout and output archives. (default: OFF)

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)

HPX_WITH_POWER_COUNTER:BOOL

Enable use of performance counters based on pwr library (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.

2.3.5 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" << std::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 <hpx/iostream.hpp>
#include "hello_world_component.hpp"

#include <iostream>

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

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/components.hpp>
#include <hpx/include/lcos.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)
    };
}} // namespace examples::server

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

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

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

#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 `$HDX_LOCATION` is the build directory or `CMAKE_INSTALL_PREFIX` you used while building *HPX*):

```
$ export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HDX_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 `$HDX_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=$PKG_CONFIG_PATH:$HDX_LOCATION/lib/pkgconfig
$ c++ -o hello_world_client hello_world_client.cpp \
```

(continues on next page)

(continued from previous page)

```
``pkg-config --cflags --libs hpx_application``\n-L${HOME}/my_hpx_libs -lhpublisher -lhpstreams
```

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"\n$ ./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`³⁸ command integrated into CMake. Following, is a Hello World component example using CMake.

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

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

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

```
# Require a recent version of cmake\ncmake_minimum_required(VERSION 3.18 FATAL_ERROR)\n\n# This project is C++ based.\nproject(your_app CXX)\n\n# Instruct cmake to find the HPX settings\nfind_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:

³⁸ https://www.cmake.org/cmake/help/latest/command/find_package.html

```
$ 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_libraries`³⁹ in your targets. The *HPX* include directories are available with the `HPX_INCLUDE_DIRS` CMake variable.

³⁹ https://www.cmake.org/cmake/help/latest/command/target_link_libraries.html

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.

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 `RPATHs`, you can include `HPX_SetFullRPATH` in your project after all libraries you want to link against have been added. Please also consult the CMake documentation [here⁴⁰](#).

⁴⁰ <https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/RPATH-handling>

Using HPX with Makefile

A basic project building with *HPX* is through creating makefiles. The process of creating one can get complex depending upon the use of cmake parameter `HPX_WITH_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 <hpx/iostream.hpp>

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

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

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

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

CXXFLAGS=-O3 -std=c++17

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

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

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

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

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

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

Important: `LINK_FLAGS` should be left empty if `HPX_WITH_HPX_MAIN` is set to OFF. Boost in the above example

is build with --layout=tagged. Actual Boost flags may vary on your build of Boost.

To build the program, type:

```
$ make
```

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

```
$ ./hello_world
```

How to build HPX components with makefile

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

hello_world_component.cpp

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

#include <iostream>

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

HDX_REGISTER_COMPONENT_MODULE()

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

HDX_REGISTER_COMPONENT(hello_world_type, hello_world)

HDX_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/components.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <hpx/include/lcos.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)
    };
}} // namespace examples::server

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

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

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

#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"
```

(continues on next page)

(continued from previous page)

```

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

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.

2.3.6 Starting the *HPX* runtime

In order to write an application that 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 blocking 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. However, it 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 that 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 that are normally available on the *HPX* command line are disabled (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 the user 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...
```

(continues on next page)

(continued from previous page)

```

    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 the ability 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 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::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 at least be 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);
}

```

(continues on next page)

(continued from previous page)

```
// ...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 the option for users to provide their 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
}
```

(continues on next page)

(continued from previous page)

```
// 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. The 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 calls 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.

2.3.7 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 that is similar to the well-known Windows INI file format⁴¹. This is a structured text format that allows users 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 equal sign '='. The name appears to the left of the equal 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. 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 '#' until the end of the line are ignored.

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

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

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

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

Built-in default configuration settings

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

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

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

The system configuration section

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

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

The `Hpx` configuration section

```
[hpx]
location = ${HPX_LOCATION:${[system.prefix]}}
component_path = ${[hpx.location]/lib/hpx:${[system.executable_prefix]/lib/hpx:${[system.
    ↪executable_prefix]}/../lib/hpx}}
master_ini_path = ${[hpx.location]/share/hpx-<version>:${[system.executable_prefix]/share/
    ↪hpx-<version>:${[system.executable_prefix]}/../share/hpx-<version>}}
ini_path = ${[hpx.master_ini_path]/ini}
os_threads = 1
cores = all
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}
trace_depth = ${HPX_TRACE_DEPTH:20}
handle_signals = ${HPX_HANDLE_SIGNALS:1}

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


Property	Description
hpx.location	This is initialized to the id of the <i>locality</i> this application instance is running on.
hpx.component_path	Duplicates are discarded. This property can refer to a list of directories separated by ':' (Linux, Android, and MacOS) or by ';' (Windows).
hpx.master_ini_paths	This is initialized to the list of default paths of the main hpx.ini configuration files. This property can refer to a list of directories separated by ':' (Linux, Android, and MacOS) or using ';' (Windows).
hpx.ini_path	This is initialized to the default path where HPX will look for more ini configuration files. This property can refer to a list of directories separated by ':' (Linux, Android, and MacOS) or using ';' (Windows).
hpx.os_threads	This setting reflects the number of OS threads used for running HPX threads. Defaults to number of detected cores (not hyperthreads/PUs).
hpx.cores	This setting reflects the number of cores used for running HPX threads. Defaults to number of detected cores (not hyperthreads/PUs).
hpx.localities	This setting reflects the number of localities the application is running on. Defaults to 1.
hpx.program_name	This setting reflects the program name of the application instance. Initialized from the command line <code>argv[0]</code> .
hpx.cmd_line	This setting reflects the actual command line used to launch this application instance.
hpx.lock_detect	This setting verifies that no locks are being held while a HPX thread is suspended. This setting is applicable only if <code>HPX_WITH_VERIFY_LOCKS</code> is set during configuration in CMake.
hpx.throw_on_lock_held	This setting causes an exception if during lock detection at least one lock is being held while a HPX thread is suspended. This setting is applicable only if <code>HPX_WITH_VERIFY_LOCKS</code> is set during configuration in CMake. This setting has no effect if <code>hpx.lock_detection=0</code> .
hpx.minimal_deadlock_timeout	This setting enables support for minimal deadlock detection for HPX threads. By default this is set to 1000ms (for Debug builds) or to 0 (for Release, RelWithDebInfo, RelMinSize builds). This setting is effective only if <code>HPX_WITH_THREAD_DEADLOCK_DETECTION</code> is set during configuration in CMake.
hpx.spinlock_deadlock_detection_limit	This setting verifies that spinlocks don't spin longer than specified using the <code>spinlock_deadlock_detection_limit</code> . This setting is applicable only if <code>HPX_WITH_SPINLOCK_DEADLOCK_DETECTION</code> is set during configuration in CMake. By default this is set to 1 (for Debug builds) or to 0 (for Release, RelWithDebInfo, RelMinSize builds).
hpx.spinlock_deadlock_timeout	This setting specifies the upper limit of the allowed number of spins that spinlocks are allowed to perform. This setting is applicable only if <code>HPX_WITH_SPINLOCK_DEADLOCK_DETECTION</code> is set during configuration in CMake. By default this is set to 1000000.
hpx.max_background_threads	This setting defines the number of threads in the scheduler, which are used to execute background tasks. 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 <code>HPX_IDLE_LOOP_COUNT_MAX</code> . This is an internal setting that 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 <code>HPX_BUSY_LOOP_COUNT_MAX</code> . This is an internal setting that you should change only if you know exactly what you are doing.
hpx.max_idle_backoff_time	This setting defines the maximum time (in milliseconds) for the scheduler to sleep after <code>hpx.max_idle_loop_count</code> iterations. This setting is applicable only if <code>HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF</code> is set during configuration in CMake. By default this is defined by the preprocessor constant <code>HPX_IDLE_BACKOFF_TIME_MAX</code> . This is an internal setting that you should change only if you know exactly what you are doing.
hpx.exception_verbose	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 <code>HPX_EXCEPTION_VERTOSITY</code> .
hpx.trace_depth	This setting defines the number of stack-levels printed in generated stack backtraces. This defaults to 20, but can be changed using the cmake <code>HPX_WITH_THREAD_BACKTRACE_DEPTH</code> configuration setting.
hpx.handle_signals	This setting defines whether HPX will register signal handlers that will print the configuration information (stack backtrace, system information, etc.) whenever a signal is raised. The default is 1. Setting this value to 0 can be useful in cases when generating a core-dump on segmentation faults or similar signals is desired. This setting has no effects on non-Linux platforms.

The hpx.threadpools configuration section

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

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

The hpx.thread_queue configuration section

Important: These are the setting control internal values used by the thread scheduling queues in the *HPX* scheduler. You should not modify these settings unless 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_STAGE: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
hpx.thread_queue. min_tasks_to_steal_pending	The value of this property defines the number of pending <i>HPX</i> threads that have to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.
hpx.thread_queue. min_tasks_to_steal_staged	The value of this property defines the number of staged <i>HPX</i> tasks that need to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.
hpx.thread_queue. min_add_new_count	The value of this property defines the minimal number of tasks to be converted into <i>HPX</i> threads whenever the thread queues for a core have run empty.
hpx.thread_queue. max_add_new_count	The value of this property defines the maximal number of tasks to be converted into <i>HPX</i> threads whenever the thread queues for a core have run empty.
hpx.thread_queue. max_delete_count	The value of this property defines the number of terminated <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
hpx.components.load_external	This entry defines whether external components will be loaded on this <i>locality</i> . This entry is normally 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
hpx.components.<component_instance_name>.name	This is the name of a component, usually the same as the second argument to the macro used while registering the component with <i>HPX_REGISTER_COMPONENT</i> . Set by the component factory.
hpx.components.<component_instance_name>.path	This is either the full path file name of the component module or the directory the component module is located in. In this case, the component module name will be derived from the property hpx.components.<component_instance_name>.name. Set by the component factory.
hpx.components.<component_instance_name>.enabled	This setting explicitly enables or disables the component. This is an optional property. HPX assumes 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 that 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 <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 that will be transferrable through the <i>parcel</i> layer. The default depends on the compile time preprocessor constant <code>HPX_PARCEL_MAX_MESSAGE_SIZE</code> (1000000000 bytes).
hpx.parcel.max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size that will be transferrable through the parcel 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.zero_copy_serialization_threshold	This property defines the threshold value (in bytes) starting at which the serialization layer will apply zero-copy optimizations for serialized entities. The default value is defined by the preprocessor constant <code>HPX_ZERO_COPY_THRESHOLD</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.
hpx.parcel.max_background_threads	This property defines how many cores should be used to perform background operations. The default is -1 (all cores).

The following settings relate to the TCP/IP parcelport.

[hpx.parcel.tcp]
enable = \${HPX_HAVE_PARCELPORT_TCP:\$[hpx.parcel.enabled]}

(continues on next page)

(continued from previous page)

```

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]
zero_copy_serialization_threshold = ${HPX_PARCEL_TCP_ZERO_COPY_SERIALIZATION_THRESHOLD}:
    ↵${[hpx.parcel.zero_copy_serialization_threshold]}
async_serialization = ${HPX_PARCEL_TCP_ASYNC_SERIALIZATION}:[hpx.parcel.async_
    ↵serialization]
parcel_pool_size = ${HPX_PARCEL_TCP_PARCEL_POOL_SIZE}:[hpx.threadpools.parcel_pool_size]
max_connections = ${HPX_PARCEL_TCP_MAX_CONNECTIONS}:[hpx.parcel.max_connections]
max_connections_per_locality = ${HPX_PARCEL_TCP_MAX_CONNECTIONS_PER_LOCALITY}:[hpx.
    ↵parcel.max_connections_per_locality]
max_message_size = ${HPX_PARCEL_TCP_MAX_MESSAGE_SIZE}:[hpx.parcel.max_message_size]
max_outbound_message_size = ${HPX_PARCEL_TCP_MAX_OUTBOUND_MESSAGE_SIZE}:[hpx.parcel.max_
    ↵outbound_message_size]
max_background_threads = ${HPX_PARCEL_TCP_MAX_BACKGROUND_THREADS}:[hpx.parcel.max_
    ↵background_threads]

```

Property	Description
hpx.parcel.tcp.enable	Enables 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.zero_copy_serialization_threshold	This property defines the threshold value (in bytes) starting at which the serialization layer applies optimizations for serialized entities. The default is the same value as set for hpx.parcel.zero_copy_serialization_threshold.
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 <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> is 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 that will be transferrable 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 that will be transferrable through the <i>parcel</i> layer. The default is taken from hpx.parcel.max_outbound_connections.
hpx.parcel.tcp.max_background_threads	This property defines how many cores should be used to perform background operations. This default is taken from hpx.parcel.max_background_threads.

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]}
zero_copy_serialization_threshold = ${HPX_PARCEL_MPI_ZERO_COPY_SERIALIZATION_THRESHOLD:
      ↳$[hpx.parcel.zero_copy_serialization_threshold]}
use_io_pool = ${HPX_HAVE_PARCEL_MPI_USE_IO_POOL:$1}
async_serialization = ${HPX_HAVE_PARCEL_MPI_ASYNC_SERIALIZATION:$[hpx.parcel.async_
      ↳serialization]}
parcel_pool_size = ${HPX_HAVE_PARCEL_MPI_PARCEL_POOL_SIZE:$[hpx.threadpools.parcel_pool_
      ↳size]}
max_connections = ${HPX_HAVE_PARCEL_MPI_MAX_CONNECTIONS:$[hpx.parcel.max_connections]}
max_connections_per_locality = ${HPX_HAVE_PARCEL_MPI_MAX_CONNECTIONS_PER_LOCALITY:$[hpx.
      ↳parcel.max_connections_per_locality]}
max_message_size = ${HPX_HAVE_PARCEL_MPI_MAX_MESSAGE_SIZE:$[hpx.parcel.max_message_
      ↳size]}
max_outbound_message_size = ${HPX_HAVE_PARCEL_MPI_MAX_OUTBOUND_MESSAGE_SIZE:$[hpx.
      ↳parcel.max_outbound_message_size]}
max_background_threads = ${HPX_PARCEL_MPI_MAX_BACKGROUND_THREADS:$[hpx.parcel.max_
      ↳background_threads]}
```

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

The hpx.agas configuration section

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

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

The `hpx.commandline` configuration section

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

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

[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
```

Note: The short options listed above are disabled by default if the application is built using `#include <hpx/hpx_main.hpp>`. See [Re-use the `main\(\)` function as the main HPX entry point](#) for more information. The rationale behind this is that in this case the user's application may handle its own command line options, since HPX passes all unknown options to `main()`. Short options like `-t` are prone to create ambiguities regarding what the application will support. Hence, the user should instead rely on the corresponding long options like `--hpx:threads` in such a case.

Property	Description
<code>hpx.commandline. aliasing</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 -a expands to: <code>--hpx:agas</code> .
<code>hpx.commandline. aliases.-c</code>	On the commandline -c expands to: <code>--hpx:console</code> .
<code>hpx.commandline. aliases.-h</code>	On the commandline -h expands to: <code>--hpx:help</code> .
<code>hpx.commandline. aliases.--help</code>	On the commandline --help expands to: <code>--hpx:help</code> .
<code>hpx.commandline. aliases.-I</code>	On the commandline -I expands to: <code>--hpx:ini</code> .
<code>hpx.commandline. aliases.-l</code>	On the commandline -l expands to: <code>--hpx:localities</code> .
<code>hpx.commandline. aliases.-p</code>	On the commandline -p expands to: <code>--hpx:app-config</code> .
<code>hpx.commandline. aliases.-q</code>	On the commandline -q expands to: <code>--hpx:queuing</code> .
<code>hpx.commandline. aliases.-r</code>	On the commandline -r expands to: <code>--hpx:run-agas-server</code> .
<code>hpx.commandline. aliases.-t</code>	On the commandline -t expands to: <code>--hpx:threads</code> .
<code>hpx.commandline. aliases.-v</code>	On the commandline -v expands to: <code>--hpx:version</code> .
<code>hpx.commandline. aliases.--version</code>	On the commandline --version expands to: <code>--hpx:version</code> .
<code>hpx.commandline. aliases.-w</code>	On the commandline -w expands to: <code>--hpx:worker</code> .
<code>hpx.commandline. aliases.-x</code>	On the commandline -x expands to: <code>--hpx:hpx</code> .
<code>hpx.commandline. aliases.-0</code>	On the commandline -0 expands to: <code>--hpx:node=0</code> .
<code>hpx.commandline. aliases.-1</code>	On the commandline -1 expands to: <code>--hpx:node=1</code> .
<code>hpx.commandline. aliases.-2</code>	On the commandline -2 expands to: <code>--hpx:node=2</code> .
<code>hpx.commandline. aliases.-3</code>	On the commandline -3 expands to: <code>--hpx:node=3</code> .
<code>hpx.commandline. aliases.-4</code>	On the commandline -4 expands to: <code>--hpx:node=4</code> .
<code>hpx.commandline. aliases.-5</code>	On the commandline -5 expands to: <code>--hpx:node=5</code> .
<code>hpx.commandline. aliases.-6</code>	On the commandline -6 expands to: <code>--hpx:node=6</code> .
<code>hpx.commandline. aliases.-7</code>	On the commandline -7 expands to: <code>--hpx:node=7</code> .
<code>hpx.commandline. aliases.-8</code>	On the commandline -8 expands to: <code>--hpx:node=8</code> .
<code>hpx.commandline. aliases.-9</code>	On the commandline -9 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 by ';' (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 by ';' (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 are 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 a 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 are responsible for creating new and discarding 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

This section assumes there is a simple application component that 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" << std::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 that define the 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. Here, one instance of the `app::server` component is created on the current *locality* and the exposed action `print_greeting_action` is invoked 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 (shown in the example below at the closing brace of the block surrounding the code):

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

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

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

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

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

In order to make sure that the application will be able to use the component `app::server`, special configuration information must be passed to *HPX*. The simplest way to allow *HPX* to ‘find’ the component is to provide special ini configuration files that 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 users 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 of 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>`.
- 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 all available messages to be logged. 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 for the use of 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 HPX, such as thread-manager, parcel layer, LCOs, etc.	<code>HPX_LOGLEVEL</code>
<code>AGAS</code>	<code>AGAS</code>	Logging output generated by the <code>AGAS</code> 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 environment 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]
<code>level = \${HPX_LOGLEVEL:0}</code>
<code>destination = \${HPX_LOGDESTINATION:console}</code>
<code>format = \${HPX_LOGFORMAT:(T%locality%/%hpxthread%.%hpxphase%/%hpxcomponent%) P%parentloc%/%hpxparent%.%hpxparentphase% %time%(\$hh:\$mm:\$ss.\$mili) [%idx%] \\n}</code>

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
<code>file(<filename>)</code>	Directs all output to a file with the given <filename>.
<code>cout</code>	Directs all output to the local standard output of the application instance on this <i>locality</i> .
<code>cerr</code>	Directs all output to the local standard error output of the application instance on this <i>locality</i> .
<code>console</code>	Directs all output to the console instance of the application. The console instance has its logging destinations configured separately.
<code>android_log</code>	Directs 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 <code>locality</code> on which the logging message was generated.
<code>hpxthread</code>	The id of the <code>HPX</code> thread generating this logging output.
<code>hpxphase</code>	The phase ⁴³ of the <code>HPX</code> thread generating this logging output.
<code>hpxcomponent</code>	The local virtual address of the component which the current <code>HPX</code> thread is accessing.
<code>parentloc</code>	The id of the <code>locality</code> where the <code>HPX</code> thread was running that initiated the current <code>HPX</code> thread. The current <code>HPX</code> thread is generating this logging output.
<code>hpxparent</code>	The id of the <code>HPX</code> thread that initiated the current <code>HPX</code> thread. The current <code>HPX</code> thread is generating this logging output.
<code>hpxparentphase</code>	The phase of the <code>HPX</code> thread when it initiated the current <code>HPX</code> thread. The current <code>HPX</code> thread is generating this logging output.
<code>time</code>	The time stamp for this logging output line as generated by the source <code>locality</code> .
<code>idx</code>	The sequence number of the logging output line as generated on the source <code>locality</code> .
<code>osthread</code>	The sequence number of the OS thread that executes the current <code>HPX</code> 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 a line break):

```
(T00000000/0000000002d46f90.01/0000000009ebc10) P-----/0000000002d46f80.02 17:49.37.
↳ 320 [000000000000004d]
    <info> [RT] successfully created component {0000000100ff0001, 0000000000030002} of
↳ type: component_barrier[7(3)]
```

The default settings for the general logging category on the console is shown here:

```
[hpx.logging.console]
level = ${HPX_LOGLEVEL:$[hpx.logging.level]}
destination = ${HPX_CONSOLE_LOGDESTINATION:file(hpx.$[system.pid].log)}
format = ${HPX_CONSOLE_LOGFORMAT:|}
```

These settings define how the logging is customized once the logging output is received by the console instance of an application. The logging level is read from the environment variable `HPX_LOGLEVEL` (as set for the console instance of the application). The level defaults to the same values as the corresponding settings in the general logging configuration shown before. The destination on the console instance is set to be a file that's name is generated based on 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.

⁴³ 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:nodedefile 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:force_ipv4

Network hostnames will be resolved to ipv4 addresses instead of using the first resolved endpoint. This is especially useful on Windows where the local hostname will resolve to an ipv6 address while remote network hostnames are commonly resolved to ipv4 addresses.

--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 include a format with a header or without any header (see option --hpx:no-csv-header). Possible values: csv (prints counter values in CSV format with full names as header), csv-short (prints counter values in CSV format with short names 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	<code>--hpx:agas</code>
-c	<code>--hpx:console</code>
-h	<code>--hpx:help</code>
-I	<code>--hpx:ini</code>
-l	<code>--hpx:localities</code>
-p	<code>--hpx:app-config</code>
-q	<code>--hpx:queueing</code>
-r	<code>--hpx:run-agas-server</code>
-t	<code>--hpx:threads</code>
-v	<code>--hpx:version</code>
-w	<code>--hpx:worker</code>
-x	<code>--hpx:hpx</code>
-0	<code>--hpx:node=0</code>
-1	<code>--hpx:node=1</code>
-2	<code>--hpx:node=2</code>
-3	<code>--hpx:node=3</code>
-4	<code>--hpx:node=4</code>
-5	<code>--hpx:node=5</code>
-6	<code>--hpx:node=6</code>
-7	<code>--hpx:node=7</code>
-8	<code>--hpx:node=8</code>
-9	<code>--hpx:node=9</code>

Note: The short options listed above are disabled by default if the application is built using `#include <hpx/hpx_main.hpp>`. See [Re-use the main\(\) function as the main HPX entry point](#) for more information. The rationale behind this is that in this case the user's application may handle its own command line options, since *HPX* passes all unknown options to `main()`. Short options like -t are prone to create ambiguities regarding what the application will support. Hence, the user should instead rely on the corresponding long options like `--hpx:threads` in such a case.

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 that 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 option 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 *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:@.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:@-3=socket:@.core:1.pu:1` is fully equivalent to its shortened form `t:@-3=s:@.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 used 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.

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. [MPI⁴²](#) 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.

2.3.8 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.

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

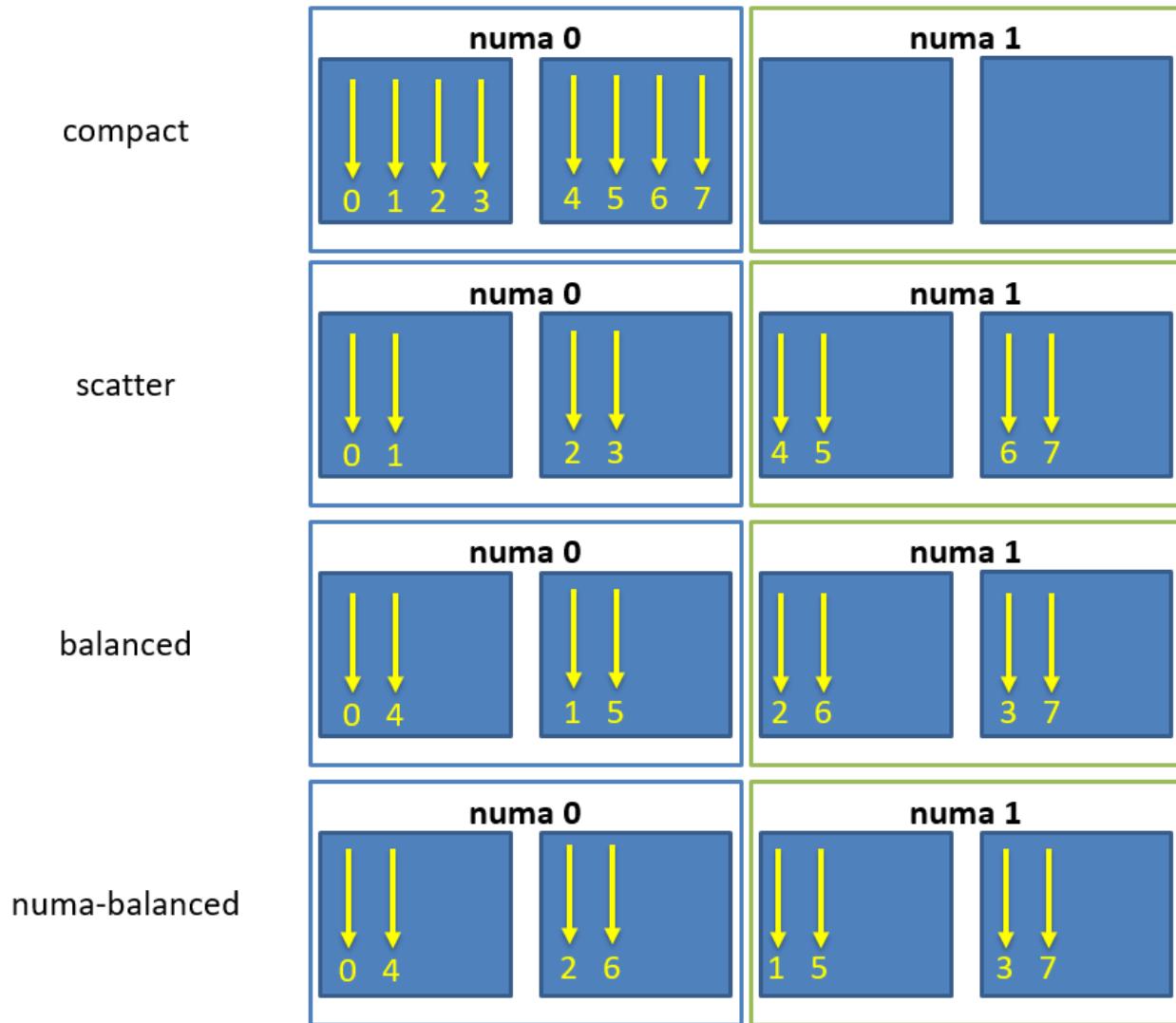


Fig. 2.7: Schematic of thread affinity type distributions.

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();
}

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>	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::future</code>	There are often situations where a <code>get()</code> call on a future may not be a blocking call, or is only a blocking call under certain circumstances. This function gives the ability to test for early completion and allows us to avoid associating a continuation, which needs to be scheduled with some non-trivial overhead and near-certain loss of cache efficiency.
<code>hpx::make_ready_future</code>	Some <code>dynfutures</code> 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 N4313⁴⁴, HPX adds functions allowing users to compose several futures in a more flexible way.

⁴⁴ <http://wg21.link/n4313>

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.	N4313 ⁴⁵ , ..._n versions are HPX 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.	HPX only
<code>hpx::when_all</code> , <code>hpx::when_all_n</code>	Asynchronously wait for all future and shared_future objects to finish.	N4313 ⁴⁶ , ..._n versions are HPX only
<code>hpx::wait_all</code> , <code>hpx::wait_all_n</code>	Synchronously wait for all future and shared_future objects to finish.	HPX only
<code>hpx::when_some</code> , <code>hpx::when_some_n</code>	Asynchronously wait for multiple future and shared_future objects to finish.	HPX only
<code>hpx::wait_some</code> , <code>hpx::wait_some_n</code>	Synchronously wait for multiple future and shared_future objects to finish.	HPX 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.	HPX only
<code>hpx::wait_each</code> , <code>hpx::wait_each_n</code>	Synchronously wait for multiple future and shared_future objects to finish and call a function for each of the future objects as soon as it becomes ready.	HPX only

High level parallel facilities

In preparation for the upcoming C++ Standards, there are currently several proposals targeting different facilities supporting parallel programming. *HPX* implements (and extends) some of those proposals. This is well aligned with our strategy to align the APIs exposed from *HPX* with current and future C++ Standards.

At this point, *HPX* implements several of the C++ Standardization working papers, most notably N4409⁴⁷ (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411⁴⁸ (Task Blocks), and N4406⁴⁹ (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 passed as

⁴⁵ <http://wg21.link/n4313>

⁴⁶ <http://wg21.link/n4313>

⁴⁷ <http://wg21.link/n4409>

⁴⁸ <http://wg21.link/n4411>

⁴⁹ <http://wg21.link/n4406>

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 cp-preference.com
<code>hpx::adjacent_find</code>	Computes the differences between adjacent elements in a range.	<hpx/algorithm.hpp>	adjacent_find ⁵⁰
<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_of ⁵¹
<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_of ⁵²
<code>hpx::count</code>	Returns the number of elements equal to a given value.	<hpx/algorithm.hpp>	count ⁵³
<code>hpx::count_if</code>	Returns the number of elements satisfying a specific criteria.	<hpx/algorithm.hpp>	count_if ⁵⁴
<code>hpx::equal</code>	Determines if two sets of elements are the same.	<hpx/algorithm.hpp>	equal ⁵⁵
<code>hpx::find</code>	Finds the first element equal to a given value.	<hpx/algorithm.hpp>	find ⁵⁶
<code>hpx::find_end</code>	Finds the last sequence of elements in a certain range.	<hpx/algorithm.hpp>	find_end ⁵⁷
<code>hpx::find_first_of</code>	Searches for any one of a set of elements.	<hpx/algorithm.hpp>	find_first_of ⁵⁸
<code>hpx::find_if</code>	Finds the first element satisfying a specific criteria.	<hpx/algorithm.hpp>	find_if ⁵⁹
<code>hpx::find_if_not</code>	Finds the first element not satisfying a specific criteria.	<hpx/algorithm.hpp>	find_if_not ⁶⁰
<code>hpx::for_each</code>	Applies a function to a range of elements.	<hpx/algorithm.hpp>	for_each ⁶¹
<code>hpx::for_each_n</code>	Applies a function to a number of elements.	<hpx/algorithm.hpp>	for_each_n ⁶²
<code>hpx::lexicographical_compare</code>	Checks if a range of values is lexicographically less than another range of values.	<hpx/algorithm.hpp>	lexicographical_compare ⁶³
<code>hpx::parallel::v1::mismatch</code>	Finds the first position where two ranges differ.	<hpx/algorithm.hpp>	mismatch ⁶⁴
<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_of ⁶⁵
<code>hpx::search</code>	Searches for a range of elements.	<hpx/algorithm.hpp>	search ⁶⁶
2.3. Manual		hpp>	117
<code>hpx::search_n</code>	Searches for a number consecutive copies of an element in a range.	<hpx/algorithm.hpp>	search_n ⁶⁷

50 http://en.cppreference.com/w/cpp/algorithm/adjacent_find
51 http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
52 http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
53 <http://en.cppreference.com/w/cpp/algorithm/count>
54 http://en.cppreference.com/w/cpp/algorithm/count_if
55 <http://en.cppreference.com/w/cpp/algorithm/equal>
56 <http://en.cppreference.com/w/cpp/algorithm/find>
57 http://en.cppreference.com/w/cpp/algorithm/find_end
58 http://en.cppreference.com/w/cpp/algorithm/find_first_of
59 http://en.cppreference.com/w/cpp/algorithm/find_if
60 http://en.cppreference.com/w/cpp/algorithm/find_if_not
61 http://en.cppreference.com/w/cpp/algorithm/for_each
62 http://en.cppreference.com/w/cpp/algorithm/for_each_n
63 http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
64 <http://en.cppreference.com/w/cpp/algorithm/mismatch>
65 http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
66 <http://en.cppreference.com/w/cpp/algorithm/search>
67 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> ⁶⁸
<code>hpx::copy_n</code>	Copies a number of elements to a new location.	<code><hpx/algorithm.hpp></code>	<code>copy_n</code> ⁶⁹
<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> ⁷⁰
<code>hpx::move</code>	Moves a range of elements to a new location.	<code><hpx/algorithm.hpp></code>	<code>move</code> ⁷¹
<code>hpx::fill</code>	Assigns a range of elements a certain value.	<code><hpx/algorithm.hpp></code>	<code>fill</code> ⁷²
<code>hpx::fill_n</code>	Assigns a value to a number of elements.	<code><hpx/algorithm.hpp></code>	<code>fill_n</code> ⁷³
<code>hpx::generate</code>	Saves the result of a function in a range.	<code><hpx/algorithm.hpp></code>	<code>generate</code> ⁷⁴
<code>hpx::generate_n</code>	Saves the result of N applications of a function.	<code><hpx/algorithm.hpp></code>	<code>generate_n</code> ⁷⁵
<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> ⁷⁶
<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> ⁷⁷
<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> ⁷⁸
<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> ⁷⁹
<code>hpx::replace</code>	Replaces all values satisfying specific criteria with another value.	<code><hpx/algorithm.hpp></code>	<code>replace</code> ⁸⁰
<code>hpx::replace_if</code>	Replaces all values satisfying specific criteria with another value.	<code><hpx/algorithm.hpp></code>	<code>replace</code> ⁸¹
<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> ⁸²
<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> ⁸³
<code>hpx::reverse</code>	Reverses the order elements in a range.	<code><hpx/algorithm.hpp></code>	<code>reverse</code> ⁸⁴
2.3. Manual		<code>hpp></code>	119
<code>hpx::reverse_copy</code>	Creates a copy of a range that is reversed.	<code><hpx/algorithm.hpp></code>	<code>reverse_copy</code> ⁸⁵
<code>hpx::parallel::v1::rotate</code>	Rotates the order of elements in a range.	<code><hpx/algorithm.hpp></code>	<code>rotate</code> ⁸⁶

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>	merge⁹⁴
<i>hpx::inplace_merge</i>	Merges two ordered ranges in-place.	<i><hpx/algorithm.hpp></i>	inplace_merge⁹⁵
<i>hpx::includes</i>	Returns true if one set is a subset of another.	<i><hpx/algorithm.hpp></i>	includes⁹⁶
<i>hpx::set_difference</i>	Computes the difference between two sets.	<i><hpx/algorithm.hpp></i>	set_difference⁹⁷
<i>hpx::set_intersection</i>	Computes the intersection of two sets.	<i><hpx/algorithm.hpp></i>	set_intersection⁹⁸
<i>hpx::set_symmetric_difference</i>	Computes the symmetric difference between two sets.	<i><hpx/algorithm.hpp></i>	set_symmetric_difference⁹⁹
<i>hpx::set_union</i>	Computes the union of two sets.	<i><hpx/algorithm.hpp></i>	set_union¹⁰⁰

⁶⁸ http://en.cppreference.com/w/cpp/algorithm/exclusive_scan

⁶⁹ http://en.cppreference.com/w/cpp/algorithm/copy_n

⁷⁰ <http://en.cppreference.com/w/cpp/algorithm/copy>

⁷¹ <http://en.cppreference.com/w/cpp/algorithm/move>

⁷² <http://en.cppreference.com/w/cpp/algorithm/fill>

⁷³ http://en.cppreference.com/w/cpp/algorithm/fill_n

⁷⁴ <http://en.cppreference.com/w/cpp/algorithm/generate>

⁷⁵ http://en.cppreference.com/w/cpp/algorithm/generate_n

⁷⁶ <http://en.cppreference.com/w/cpp/algorithm/remove>

⁷⁷ <http://en.cppreference.com/w/cpp/algorithm/remove>

⁷⁸ http://en.cppreference.com/w/cpp/algorithm/remove_copy

⁷⁹ http://en.cppreference.com/w/cpp/algorithm/remove_copy

⁸⁰ <http://en.cppreference.com/w/cpp/algorithm/replace>

⁸¹ <http://en.cppreference.com/w/cpp/algorithm/replace>

⁸² http://en.cppreference.com/w/cpp/algorithm/replace_copy

⁸³ http://en.cppreference.com/w/cpp/algorithm/replace_copy

⁸⁴ <http://en.cppreference.com/w/cpp/algorithm/reverse>

⁸⁵ http://en.cppreference.com/w/cpp/algorithm/reverse_copy

⁸⁶ <http://en.cppreference.com/w/cpp/algorithm/rotate>

⁸⁷ http://en.cppreference.com/w/cpp/algorithm/rotate_copy

⁸⁸ http://en.cppreference.com/w/cpp/algorithm/shift_left

⁸⁹ http://en.cppreference.com/w/cpp/algorithm/shift_right

⁹⁰ http://en.cppreference.com/w/cpp/algorithm/swap_ranges

⁹¹ <http://en.cppreference.com/w/cpp/algorithm/transform>

⁹² <http://en.cppreference.com/w/cpp/algorithm/unique>

⁹³ http://en.cppreference.com/w/cpp/algorithm/unique_copy

⁹⁴ <http://en.cppreference.com/w/cpp/algorithm/merge>

⁹⁵ http://en.cppreference.com/w/cpp/algorithm/inplace_merge

⁹⁶ <http://en.cppreference.com/w/cpp/algorithm/includes>

⁹⁷ http://en.cppreference.com/w/cpp/algorithm/set_difference

⁹⁸ http://en.cppreference.com/w/cpp/algorithm/set_intersection

⁹⁹ http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference

¹⁰⁰ 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 true if the range is max heap.	<hpx/algorithm.hpp>	is_heap ¹⁰¹
<code>hpx::is_heap_until</code>	Returns the first element that breaks a max heap.	<hpx/algorithm.hpp>	is_heap_until ¹⁰²
<code>hpx::make_heap</code>	Constructs a max heap in the range [first, last).	<hpx/algorithm.hpp>	make_heap ¹⁰³

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>	max_element ¹⁰⁴
<code>hpx::min_element</code>	Returns the smallest element in a range.	<hpx/algorithm.hpp>	min_element ¹⁰⁵
<code>hpx::minmax_element</code>	Returns the smallest and the largest element in a range.	<hpx/algorithm.hpp>	minmax_element ¹⁰⁶

¹⁰¹ http://en.cppreference.com/w/cpp/algorithm/is_heap¹⁰² http://en.cppreference.com/w/cpp/algorithm/is_heap_until¹⁰³ http://en.cppreference.com/w/cpp/algorithm/make_heap¹⁰⁴ http://en.cppreference.com/w/cpp/algorithm/max_element¹⁰⁵ http://en.cppreference.com/w/cpp/algorithm/min_element¹⁰⁶ 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 ¹⁰⁷
<code>hpx::exclusive_scan</code>	Does an exclusive parallel scan over a range of elements.	<code><hpx/numeric.hpp></code>	exclusive_scan ¹⁰⁸
<code>hpx::reduce</code>	Sums up a range of elements.	<code><hpx/numeric.hpp></code>	reduce ¹⁰⁹
<code>hpx::inclusive_scan</code>	Does an inclusive parallel scan over a range of elements.	<code><hpx/algorithm.hpp></code>	inclusive_scan ¹¹⁰
<code>hpx::parallel_inclusive_scan</code>	Performs an inclusive scan on consecutive elements with matching keys, with a reduction to output only the final sum for each key. The key sequence {1, 1, 1, 2, 3, 3, 3, 3, 1} and value sequence {2, 3, 4, 5, 6, 7, 8, 9, 10} would be reduced to keys={1, 2, 3, 1}, values={9, 5, 30, 10}.	<code><hpx/numeric.hpp></code>	
<code>hpx::transform_reduce</code>	Sums up a range of elements after applying a function. Also, accumulates the inner products of two input ranges.	<code><hpx/numeric.hpp></code>	transform_reduce ¹¹¹
<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 ¹¹²
<code>hpx::parallel_transform_inclusive_scan</code>	Does an inclusive parallel scan over a range of elements after applying a function.	<code><hpx/numeric.hpp></code>	transform_exclusive_scan ¹¹³

¹⁰⁷ http://en.cppreference.com/w/cpp/algorithm/adjacent_difference

¹⁰⁸ http://en.cppreference.com/w/cpp/algorithm/exclusive_scan

¹⁰⁹ <http://en.cppreference.com/w/cpp/algorithm/reduce>

¹¹⁰ http://en.cppreference.com/w/cpp/algorithm/inclusive_scan

¹¹¹ http://en.cppreference.com/w/cpp/algorithm/transform_reduce

¹¹² http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan

¹¹³ 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>	destroy¹¹⁴
<i>hpx::destroy_n</i>	Destroys a range of objects.	<i><hpx/memory.hpp></i>	destroy_n¹¹⁵
<i>hpx::uninitialized_copy</i>	Copies a range of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_copy¹¹⁶
<i>hpx::uninitialized_copy_n</i>	Copies a number of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_copy_n¹¹⁷
<i>hpx::uninitialized_default</i>	Copies a range of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_default_construct¹¹⁸
<i>hpx::uninitialized_default_n</i>	Copies a number of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_default_construct_n¹¹⁹
<i>hpx::uninitialized_fill</i>	Copies an object to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_fill¹²⁰
<i>hpx::uninitialized_fill_n</i>	Copies an object to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_fill_n¹²¹
<i>hpx::uninitialized_move</i>	Moves a range of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_move¹²²
<i>hpx::uninitialized_move_n</i>	Moves a number of objects to an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_move_n¹²³
<i>hpx::uninitialized_value_construct</i>	Constructs objects in an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_value_construct¹²⁴
<i>hpx::uninitialized_value_construct_n</i>	Constructs objects in an uninitialized area of memory.	<i><hpx/memory.hpp></i>	uninitialized_value_construct_n¹²⁵

¹¹⁴ <http://en.cppreference.com/w/cpp/memory/destroy>

¹¹⁵ http://en.cppreference.com/w/cpp/memory/destroy_n

¹¹⁶ http://en.cppreference.com/w/cpp/memory/uninitialized_copy

¹¹⁷ http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n

¹¹⁸ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct

¹¹⁹ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n

¹²⁰ http://en.cppreference.com/w/cpp/memory/uninitialized_fill

¹²¹ http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n

¹²² http://en.cppreference.com/w/cpp/memory/uninitialized_move

¹²³ http://en.cppreference.com/w/cpp/memory/uninitialized_move_n

¹²⁴ http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct

¹²⁵ 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::experimental::for_loop</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/algorithm.hpp></code>
<code>hpx::experimental::for_loop</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/algorithm.hpp></code>
<code>hpx::experimental::for_loop</code>	Implements loop functionality over a range specified by integral or iterator bounds.	<code><hpx/algorithm.hpp></code>
<code>hpx::experimental::for_loop</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 N4411¹²⁶ are based on the `task_block` concept that is a part of the common subset of the Microsoft Parallel Patterns Library (PPL)¹²⁷ and the Intel Threading Building Blocks (TBB)¹²⁸ 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++¹²⁹ and `async` and `finish` from X10¹³⁰. 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;
}
```

¹²⁶ <http://wg21.link/n4411>

¹²⁷ <https://msdn.microsoft.com/en-us/library/dd492418.aspx>

¹²⁸ <https://www.threadingbuildingblocks.org/>

¹²⁹ <https://software.intel.com/en-us/articles/intel-cilk-plus/>

¹³⁰ <https://x10-lang.org/>

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

The `task_block` function delineates a region in a program code potentially containing invocations of threads spawned by the `run` member function of the `task_block` class. The `run` function spawns an *HPX* thread, a unit of work that is allowed to execute in parallel with respect to the caller. Any parallel tasks spawned by `run` within the task block are joined back to a single thread of execution at the end of the `define_task_block`. `run` takes a user-provided function object `f` and starts it asynchronously—i.e., it may return before the execution of `f` completes. The *HPX* scheduler may choose to run `f` immediately or delay running `f` until compute resources become available.

A `task_block` can be constructed only by `define_task_block` because it has no public constructors. Thus, `run` can be invoked directly or indirectly only from a user-provided function passed to `define_task_block`:

```
void g();

void f(task_block<>& tr)
{
    tr.run(g);           // OK, invoked from within task_block in h
}

void h()
{
    define_task_block(f);
}

int main()
{
    task_block<> tr;    // Error: no public constructor
    tr.run(g);           // No way to call run outside of a define_task_block
    return 0;
}
```

Extensions for task blocks

Using execution policies with task blocks

HPX implements some extensions for `task_block` beyond the actual standards proposal N4411¹³¹. 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
```

(continues on next page)

¹³¹ <http://wg21.link/n4411>

(continued from previous page)

```
[&](task_block<>& tb) {
    if (n->left)
        tb.run([&] { left = traverse(n->left, compute); });
    if (n->right)
        tb.run([&] { right = traverse(n->right, compute); });
};

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

This also causes the `hpx::parallel::v2::task_block` object to be a template in our implementation. The template argument is the type of the execution policy used to create the task block. The template argument defaults to `hpx::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, // execution::parallel_policy
        [&](auto& tb) {
            if (n->left)
            {
                // use explicitly specified executor to run this task
                tb.run(my_executor(), [&] { left = traverse(n->left, compute); });
            }
            if (n->right)
            {
                // use the executor associated with the par execution policy
                tb.run([&] { right = traverse(n->right, compute); });
            }
        });
    return compute(n) + left + right;
}
```

HPX still supports calling `hpx::parallel::v2::task_block::run` without an explicit executor object. In this case the task will be run using the executor associated with the execution policy that was used to call `hpx::parallel::v2::define_task_block`.

2.3.9 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 exposes 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`: retrieves the global address of the *locality* this function is called on.
- `hpx::find_all_localities`: retrieves the global addresses of all localities available to this application (including the *locality* the function is being called on).
- `hpx::find_remote_localities`: retrieves 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`: retrieves the number of localities available to this application.
- `hpx::find_locality`: retrieves the global address of any *locality* supporting the given component type.
- `hpx::get_colocation_id`: retrieves 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_`: Creates 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 used 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 in 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);
```

While these macro invocations are a bit more complex than those for simple global functions, they should still be 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. 2.8 below shows an overview of the main API exposed by *HPX*. This shows the function invocation syntax as defined by the C++ language (dark gray), the additional invocation syntax as provided through C++ Standard Library features (medium gray), and the extensions added by *HPX* (light gray) where:

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

R f(p...)	Synchronous Execution (returns R)	Asynchronous Execution (returns <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` asynchronously triggers a chain of functions without the need for the execution flow ‘to come back’ to the invocation site. Each of the asynchronous functions can be executed on a different *locality*.

Applying an action with a continuation and with synchronization

This method is very similar to the method described in section [Applying an action asynchronously with synchronization](#). In addition to what `hpx::async` can do, the functions `hpx::async_continue` takes an additional function argument. This function will be called as the continuation of the executed action. It is expected to perform additional operations and to make sure that a result is returned to the original invocation site. This method chains operations asynchronously by providing a continuation operation which is automatically executed once the first action has finished executing.

As an example we chain two actions, where the result of the first action is forwarded to the second action and the result of the second action is sent back to the original invocation site:

```
// first action
std::int32_t action1(std::int32_t i)
{
    return i+1;
}
HPX_PLAIN_ACTION(action1);      // defines action1_type

// second action
std::int32_t action2(std::int32_t i)
{
    return i*2;
}
HPX_PLAIN_ACTION(action2);      // defines action2_type

// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;      // define an instance of 'action1_type'
action2_type act2;      // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1, hpx::make_continuation(act2),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n";  // will print: 86 ((42 + 1) * 2)
```

By default, the continuation is executed on the same *locality* as `hpx::async_continue` is invoked from. If you want to specify the *locality* where the continuation should be executed, the code above has to be written as:

```
// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;      // define an instance of 'action1_type'
action2_type act2;      // define an instance of 'action2_type'
hpx::future<int> f =
```

(continues on next page)

(continued from previous page)

```
hpx::async_continue(act1, hpx::make_continuation(act2, hpx::find_here()),
    hpx::find_here(), 42);
hpx::cout << f.get() << "\n"; // will print: 86 ((42 + 1) * 2)
```

Similarly, it is possible to chain more than 2 operations:

```
action1_type act1;      // define an instance of 'action1_type'
action2_type act2;      // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1,
        hpx::make_continuation(act2, hpx::make_continuation(act1)),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n"; // will print: 87 ((42 + 1) * 2 + 1)
```

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

```
struct continuation
{
    template <typename Result>
    void operator()(hpx::id_type id, Result&& result) const
    {
        ...
    }
};
```

where the parameters passed to the overloaded function `operator()` are:

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

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

Action error handling

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

Important: Exceptions thrown during asynchronous execution can be transferred back to the invoking thread only for the synchronous and the asynchronous case with synchronization. Like with any other unhandled exception, any exception thrown during the execution of an asynchronous action *without* synchronization will result in calling `hpx::terminate` causing the running application to exit immediately.

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

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

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

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

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

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

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

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

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

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

Writing components

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

Defining components

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

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

```
// header file some_component.hpp

#include <hpx/include/components.hpp>

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

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

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

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

For instance:

```
// source file some_component.cpp
```

(continues on next page)

(continued from previous page)

```
#include "some_component.hpp"

// The following code generates all necessary boiler plate to enable the
// remote creation of 'app::some_component' instances with 'hpx::new<>()'
//
using some_component = app::some_component;
using some_component_type = hpx::components::component<some_component>;

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

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

Defining client side representation classes

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

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

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

namespace app
{
    // Define a client side representation type for the component type
    // 'some_component' defined in the previous section.
    //
    struct some_component_client
        : hpx::components::client_base<some_component_client, some_component>
    {
        using base_type = hpx::components::client_base<
            some_component_client, some_component>;
        some_component_client(hpx::future<hpx::id_type> && id)
            : base_type(std::move(id))
        {}
        hpx::future<int> some_member_function(std::string const& s)
        {
            some_component::some_member_action act;
```

(continues on next page)

(continued from previous page)

```

        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), ...);

```

(continues on next page)

(continued from previous page)

```
// 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 );

// 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"
            //
        });

```

(continues on next page)

(continued from previous page)

```
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_vector	Dynamic segmented contiguous array.	<hpx/include/partitioned_vector.hpp>	vector ¹³²

Table 2.22: Unordered associative containers

Name	Description	In header	Class page at cppreference.com
hpx::unordered_map	Segmented collection of key-value pairs, hashed by keys, keys are unique.	<hpx/include/unordered_map.hpp>	unordered_map ¹³³

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
```

(continues on next page)

¹³² <http://en.cppreference.com/w/cpp/container/vector>

¹³³ http://en.cppreference.com/w/cpp/container/unordered_map

(continued from previous page)

```
//  
HPX_REGISTER_PARTITIONED_VECTOR(double);  
  
using iterator = hpx::partitioned_vector<T>::iterator;  
using traits = hpx::traits::segmented_iterator_traits<iterator>;  
  
hpx::partitioned_vector<T> v;  
std::size_t count = 0;  
  
auto seg_begin = traits::segment(v.begin());  
auto seg_end = traits::segment(v.end());  
  
// Iterate over segments  
for (auto seg_it = seg_begin; seg_it != seg_end; ++seg_it)  
{  
    auto loc_begin = traits::begin(seg_it);  
    auto loc_end = traits::end(seg_it);  
  
    // Iterate over elements inside segments  
    for (auto lit = loc_begin; lit != loc_end; ++lit, ++count)  
    {  
        *lit = count;  
    }  
}
```

Which is equivalent to:

```
hpx::partitioned_vector<T> v;  
std::size_t count = 0;  
  
auto begin = v.begin();  
auto end = v.end();  
  
for (auto it = begin; it != end; ++it, ++count)  
{  
    *it = count;  
}
```

Using views

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

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

Preface: Why SPMD?

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

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

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

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */)
{
    // Parallel section

    /* Do some code */
}
HPX_PLAIN_ACTION(bulk_function, bulk_action);
```

Note: In the following paragraphs, we will use the term “image” several times. An image is defined as a lightweight process whose entry point is a function provided by the user. It’s an “image of the function”.

The `spmd_block` class contains the following methods:

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

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

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

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */)
{
    std::size_t num_images = block.get_num_images();
    std::size_t this_image = block.this_image();
    std::size_t images_per_locality = block.images_per_locality();

    /* Do some code */

    // Synchronize all images in the team
    block.sync_all();

    /* Do some code */

    // Synchronize image 0 and image 1
    block.sync_images(0,1);

    /* Do some code */

    std::vector<std::size_t> vec_images = {2,3,4};

    // Synchronize images 2, 3 and 4
    block.sync_images(vec_images);
```

(continues on next page)

(continued from previous page)

```

// 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>;
```

(continues on next page)

(continued from previous page)

```

/* 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);

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();

```

(continues on next page)

(continued from previous page)

```

auto vv1_it = vv1.cbegin();

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

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

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

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

```

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

```

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

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

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

/* Do some code */

Vec v;

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

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

    // Instantiate the local view from the view

```

(continues on next page)

(continued from previous page)

```

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(
        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;
    }
}

```

(continues on next page)

(continued from previous page)

}

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

Note that the “last dimension size” cannot be set by the user. It only accepts the `constexpr` variable `hpx::container::placeholders::_`. This size, which is considered private, is equal to the number of current images (value returned by `block.get_num_images()`).

Note: An important constraint to remember about coarray objects is that all segments sharing the same “last dimension index” are located in the same image.

Using co-arrays

The member functions owned by the coarray objects are exactly the same as those of spmd multidimensional views. These are:

- * Subscript-based operations
- * Iterator-based operations

However, one additional functionality is provided. Knowing that the element `a(i,j,k)` is in the memory of the `k`th image, the use of local subscripts is possible.

Note: For spmd multidimensional views, subscripts are only global as it still involves potential remote data transfers.

Here is an example of using local subscripts:

```
#include <hpx/components/containers/coarray/coarray.hpp>
#include <hpx/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++)

```

(continues on next page)

(continued from previous page)

```

for (std::size_t i = 0; i<height; i++)
{
    // Local write operation performed via the use of local subscript
    a(i,j,...) = std::vector<double>(elt_size, idx);
    idx++;
}

block.sync_all();
}

```

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

2.3.10 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 documentation¹³⁴ 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
```

¹³⁴ <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:node=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=
```

(continues on next page)

(continued from previous page)

```
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.

2.3.11 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`.

2.3.12 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 <i>locality#*</i> or <i>worker-thread#*</i> . 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

Command 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</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</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</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</code>	Lists the description of all registered performance counters.
<code>--hpx:print</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: <code>csv</code> (prints counter values in CSV format with full names as header) <code>csv-short</code> (prints counter values in CSV format with shortnames provided with <code>--hpx:print-counter</code> as <code>--hpx:print-counter shortname,full-countername</code>).
<code>--hpx:print</code>	Prints the performance counter(s) specified with <code>--hpx:print-counter</code> and <code>csv</code> or <code>csv-short</code> format specified with <code>--hpx:print-counter-format</code> without header.
<code>--hpx:print</code>	Prints the performance counter(s) specified with <code>--hpx:print-counter</code> (or arg <code>--hpx:print-counter-reset</code>) at the given point in time. Possible argument values: <code>startup</code> , <code>shutdown</code> (default), <code>noshutdown</code> .
<code>--hpx:reset</code>	Resets the performance counter(s) specified with <code>--hpx:print-counter</code> after they have been evaluated.
<code>--hpx:print</code>	Appends the type description to generated output.
<code>--hpx:print</code>	Prints locally 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_type::raw version: 1.0.0
```

```
fullname: /agas/count/bind helptext: returns the number of invocations of the
AGAS service 'bind' type: counter_type::raw version: 1.0.0
```

```
fullname: /agas/count/bind_gid helptext: returns the number of invocations of
the AGAS service 'bind_gid' type: counter_type::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
```

(continues on next page)

(continued from previous page)

```
/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() << std::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");
```

(continues on next page)

(continued from previous page)

```
hpx::future<int> result = count.get_value<int>();
hpx::cout << result.get() << std::endl;
```

Providing performance counter data

HPX offers several ways by which you may provide your own data as a performance counter. This has the benefit of exposing additional, possibly application-specific information using the existing Performance Counter framework, unifying the process of gathering data about your application.

An application that wants to provide counter data can implement a performance counter to provide the data. When a consumer queries performance data, the HPX runtime system calls the provider to collect the data. The runtime system uses an internal registry to determine which provider to call.

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

Exposing performance counter data using a simple function

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

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

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

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

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

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

This example function exposes a linearly-increasing value as our performance data. The value is incremented on each invocation, i.e., each time a consumer requests the counter data of this performance counter.

The next step in exposing this counter to the runtime system is to register the function as a new raw counter type using the HPX API function `hpx::performance_counters::install_counter_type`. A counter type represents certain common characteristics of counters, like their counter type name and any associated description information. The following snippet shows an example of how to register the function `some_performance_data`, which is shown above, for a counter type named "/test/data". This registration has to be executed before any consumer instantiates, and queries an instance of this counter type:

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

void register_counter_type()
{
    // Call the HPX API function to register the counter type.
    hpx::performance_counters::install_counter_type(
        "/test/data",                                // counter type name
        &some_performance_data,                      // function providing counter
→ data
        "returns a linearly increasing counter value" // description text (optional)
        ""                                         // unit of measure (optional)
    );
}
```

Now it is possible to instantiate a new counter instance based on the naming scheme "/test{locality#*/total}/data" where * is a zero-based integer index identifying the *locality* for which the counter instance should be accessed. The function `hpx::performance_counters::install_counter_type` enables users to instantiate exactly one counter instance for each *locality*. Repeated requests to instantiate such a counter will return the same instance, i.e., the instance created for the first request.

If this counter needs to be accessed using the standard *HPX* command line options, the registration has to be performed during application startup, before `hpx_main` is executed. The best way to achieve this is to register an *HPX* startup function using the API function `hpx::register_startup_function` before calling `hpx::init` to initialize the runtime system:

```
int main(int argc, char* argv[])
{
    // By registering the counter type we make it available to any consumer
    // who creates and queries an instance of the type "/test/data".
    //
    // This registration should be performed during startup. The
    // function 'register_counter_type' should be executed as an HPX thread right
    // before hpx_main is executed.
    hpx::register_startup_function(&register_counter_type);

    // Initialize and run HPX.
    return hpx::init(argc, argv);
}
```

Please see the code in `simplest_performance_counter.cpp` for a full example demonstrating this functionality.

Implementing a full performance counter

Sometimes, the simple way of exposing a single value as a performance counter is not sufficient. For that reason, *HPX* provides a means of implementing full performance counters which support:

- Retrieving the descriptive information about the performance counter
- Retrieving the current counter value
- Resetting the performance counter (value)
- Starting the performance counter
- Stopping the performance counter

- Setting the (initial) value of the performance counter

Every full performance counter will implement a predefined interface:

```
// Copyright (c) 2007-2020 Hartmut Kaiser
//
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#include <hpx/async_base/launch_policy.hpp>
#include <hpx/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>

#include <hpx/config/warnings_prefix.hpp>

////////////////////////////////////////////////////////////////
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))
    {
    }
}
```

(continues on next page)

(continued from previous page)

```

}

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(
    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)
{
}

```

(continues on next page)

(continued from previous page)

```

        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,
            hpx::bind_front(&performance_counter::extract_value<T>));
    }
    template <typename T>
    T get_value(
        launch::sync_policy, bool reset = false, error_code& ec = throws)
    {
        return get_counter_value(launch::sync, reset).get_value<T>(ec);
    }

    template <typename T>
    future<T> get_value() const
    {
        return get_counter_value().then(hpx::launch::sync,
            hpx::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

#include <hpx/config/warnings_suffix.hpp>

```

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>

```

(continues on next page)

(continued from previous page)

```

#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:
        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)));
        }
    };
}}

```

(continues on next page)

(continued from previous page)

```
}} // namespace hpx::performance_counters
```

The single template parameter is expected to receive the type of the derived class implementing the performance counter. In the Sine example this looks like:

```
// Copyright (c) 2007-2012 Hartmut Kaiser
//
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#ifndef 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>
    //{
    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();
}}
```

(continues on next page)

(continued from previous page)

```
private:
    typedef hpx::spinlock mutex_type;

    mutable mutex_type mtx_;
    double current_value_;
    std::uint64_t evaluated_at_;

    hpx::util::interval_timer timer_;
};

}}}// namespace performance_counters::sine::server
#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 article](#)¹³⁵. This base class itself is derived from the `performance_counter` interface described above.

Additionally, a full performance counter implementation not only exposes the actual value but also provides information about:

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

Existing HPX performance counters

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

¹³⁵ http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Table 2.27: AGAS performance counters

Counter type	Counter instance formatting	Description	Parameters
/agas/count/<agas_service> ?? where: <agas_service> is one of the following: <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	<agas_instance>/total where: <agas_instance> is the name of the AGAS service to query. Currently, this value will be locality#0 where 0 is the root <i>locality</i> (the id of the locality hosting the AGAS service). 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, bin, resolve, unbind, and iterate_names (only the primary and symbol AGAS service components live on all localities, whereas all other AGAS services are available on locality#0 only).	None	Returns the total number of invocations of the specified AGAS service since its creation.
/agas/<agas_service_category>/count ?? where: <agas_service_category> is one of the following: primary, locality, component or symbol	<agas_instance>/total where: <agas_instance> is the name of the AGAS service to query. Currently, this value will be locality#0 where 0 is the root <i>locality</i> (the id of the <i>locality</i> 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).	None	Returns the overall total number of invocations of all AGAS services provided by the given AGAS service category since its creation.
agas/time/<agas_service> ?? where: <agas_service> is one of the following: <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,	<agas_instance>/total where: <agas_instance> is the name of the AGAS service to query. Currently, this value will be locality#0 where 0 is the root <i>locality</i> (the id of the <i>locality</i> hosting the AGAS service). 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, bin, resolve, unbind, and iterate_names (only the primary and symbol AGAS service components live on all localities, whereas all other AGAS services are available on locality#0 only).	None	Returns the overall execution time of the specified AGAS service since its creation (in nanoseconds).

Table 2.28: 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. sent or received) for the specified <connection_type>. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the HPX core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the HPX core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see CMake variables used to configure HPX for more details.	None
/data/time/ <connection_type> <operation> ?? where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi	locality#*/ total where: * is the <i>locality</i> id of the <i>locality</i> the total transmission time should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the total time (in nanoseconds) between the start of each asynchronous transmission operation and the end of the corresponding operation for the specified <connection_type> the given <i>locality</i> (see <operation>, e.g. sent or received). The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the HPX core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the HPX core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see CMake variables used to configure HPX for more details.	None
/serialize/ count/ <connection_type> <operation> ?? where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi	locality#*/ total where: * is the <i>locality</i> id of the <i>locality</i> the overall number of transmitted bytes should be queried for. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall number of bytes transferred (see <operation>, e.g. sent or received possibly compressed) for the specified <connection_type> by the given <i>locality</i> . The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the HPX core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the HPX core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI.	If the configuration option -DHPX_WITH_PARCELPORT_ACTION 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.
174	number identifying the <i>locality</i> .	Please see CMake variables Chapter 2: What's so special about HPX? for more details.	
/serialize/	locality#*/	Returns the overall time spent performing outgoing data se-	If the configura-

Table 2.29: Thread manager performance counters

Counter type	Counter instance formatting	Description	Parameters
/threads/count/ cumulative ??	<p>locality#*/total or 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 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 thread count is returned.</p> <p>idle-loop counter should be queried for.</p> <p>worker-thread#* is defining 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>	<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 all worker threads separately. This counter is available only if the configuration <code>HPX_THREADS_CUMULATIVE_COUNTS</code> is set to ON (default: ON).</p>	None

continues on next page

¹³⁶ A message can potentially consist of more than one *parcel*.

Table 2.29 – continued from previous page

/threads/time/ average ??	<p><code>locality#*/total</code> or <code>locality#*/</code> <code>worker-thread#*</code> or <code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <code>locality</code> for which the average time spent executing one <code>HPX</code>-thread should be queried for. The <code>locality</code> id (given by <code>*</code> is a (zero based) number identifying the <code>locality</code>.</p> <p><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> is defining the worker thread for which the average time spent executing one <code>HPX</code>-thread should be queried for. The worker thread number (given by the <code>*</code> 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 <code>HPX</code>-thread on the given <code>locality</code> since application start. If the instance name is <code>total</code> the counter returns the average time spent executing one <code>HPX</code>-thread for all worker threads (cores) on that <code>locality</code>. If the instance name is <code>worker-thread#*</code> the counter will return the average time spent executing one <code>HPX</code>-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><code>locality#*/total</code> or <code>locality#*/ worker-thread#*</code> or <code>locality#*/pool#*/ worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <code>locality</code> for which the average overhead spent executing one <code>HPX</code>-thread should be queried for. The <code>locality</code> id (given by <code>*</code> is a (zero based) number identifying the <code>locality</code>.</p> <p><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> is defining the worker thread for which the average overhead spent executing one <code>HPX</code>-thread should be queried for. The worker thread number (given by the <code>*</code> 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 <code>HPX</code>-thread on the given <code>locality</code> since application start. If the instance name is <code>total</code> the counter returns the average time spent on overhead while executing one <code>HPX</code>-thread for all worker threads (cores) on that <code>locality</code>. If the instance name is <code>worker-thread#*</code> the counter will return the average time spent on overhead executing one <code>HPX</code>-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><code>locality#*/total</code> or <code>locality#*/ worker-thread#*</code> or <code>locality#*/pool#*/ worker-thread#*</code></p> <p>where: <code>locality#*</code> is defining the <code>locality</code> for which the overall number of executed HPX-thread phases (invocations) should be queried for. The <code>locality</code> id (given by <code>*</code> is a (zero based) number identifying the <code>locality</code>. <code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for. <code>worker-thread#*</code> 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 <code>*</code> 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 HPX-thread phases (invocations) on the given <code>locality</code> since application start. If the instance name is <code>total</code> the counter returns the accumulated number of executed HPX-thread phases (invocations) for all worker threads (cores) on that <code>locality</code> . If the instance name is <code>worker-thread#*</code> 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].	None
--	---	--	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ average-phase ??	<p><code>locality#*/total</code> or <code>locality#*/ worker-thread#*</code> or <code>locality#*/pool#*/ worker-thread#*</code> where: <code>locality#*</code> 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>. <code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for. <code>worker-thread#*</code> 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 <code>total</code> 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 <code>worker-thread#*</code> 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 or locality#*/ worker-thread#* or locality#*/pool#*/ worker-thread#* where: locality#* is defining the <i>locality</i> for which the average time overhead executing one HPX-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 HPX-thread phase (invocation) should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <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 HPX-thread phase (invocation) 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 HPX-thread phase (invocation) 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 HPX-thread phase for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_THREAD_CUMULATIVE_COUNTS (default: ON) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].	None
--	---	---	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ overall ??	<p><code>locality#*/total</code> or <code>locality#*/</code> <code>worker-thread#*</code></p> <p>or</p> <p><code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <code>locality</code> for which the overall time spent running the scheduler should be queried for. The <code>locality</code> id (given by * is a (zero based) number identifying the <code>locality</code>.</p> <p><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> 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 <code>locality</code> since application start. If the instance name is <code>total</code> the counter returns the overall time spent running the scheduler for all worker threads (cores) on that <code>locality</code>. If the instance name is <code>worker-thread#*</code> 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 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/ cumulative ??	<p><code>locality#*/total</code> or <code>locality#*/ worker-thread#*</code> or <code>locality#*/pool#*/ worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <i>locality</i> for which the overall time spent executing all 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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> is defining the worker thread for which the overall time spent executing all HPX-threads should be queried for. The worker thread number (given by * 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 HPX-threads on the given <i>locality</i> since application start. If the instance name is <code>total</code> the counter returns the overall time spent executing all HPX-threads for all worker threads (cores) on that <i>locality</i>. If the instance name is <code>worker-thread#*</code> the counter will return the overall time spent executing all HPX-threads for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_THREAD_MAINTAIN_IDLE_RATES</code> are set to ON (default: OFF).</p>	None
------------------------------------	---	--	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/ cumulative-overheads ??	<p><code>locality#*/total</code> or <code>locality#*/ worker-thread#*</code> or <code>locality#*/pool#*/ worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <i>locality</i> for which the overall overhead time incurred by executing all HPX-threads should be queried for. The <i>locality</i> id (given by <code>*</code>) is a (zero based) number identifying the <i>locality</i>.</p> <p><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> is defining the worker thread for which the overall overhead time incurred by executing all HPX-threads should be queried for. The worker thread number (given by the <code>*</code>) 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 overhead time incurred executing all HPX-threads on the given <i>locality</i> since application start. If the instance name is <code>total</code> the counter returns the overall overhead time incurred executing all HPX-threads for all worker threads (cores) on that <i>locality</i>. If the instance name is <code>worker-thread#*</code> the counter will return the overall overhead time incurred executing all HPX-threads for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_THREAD_MAINTAIN_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].</p>	None
--	---	---	------

continues on next page

Table 2.29 – continued from previous page

<p><code>threads/count/</code> <code>instantaneous/</code> <code><thread-state></code> <code>??</code></p> <p>where: <code><thread-state></code> is one of the following: <code>all</code>, <code>active</code>, <code>pending</code>, <code>suspended</code>, <code>terminated</code>, <code>staged</code></p>	<p><code>locality#/*/total</code> or <code>locality#/*/</code> <code>worker-thread#*</code> or <code>locality#/*/pool#/*/</code> <code>worker-thread#*</code> where: <code>locality#*</code> is defining the <code>locality</code> for which the current number of threads with the given state should be queried for. The <code>locality</code> id (given by <code>*</code> is a (zero based) number identifying the <code>locality</code>. <code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for. <code>worker-thread#*</code> 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 <code>*</code> 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. The <code>staged</code> thread state refers to registered tasks before they are converted to thread objects.</p>	<p>Returns the current number of HPX-threads having the given thread state on the given <code>locality</code>. If the instance name is <code>total</code> the counter returns the current number of HPX-threads of the given state for all worker threads (cores) on that <code>locality</code>. If the instance name is <code>worker-thread#*</code> the counter will return the current number of HPX-threads in the given state for all worker threads separately.</p>	<p>None</p>
---	---	---	-------------

continues on next page

Table 2.29 – continued from previous page

<p><code>threads/wait-time/ <thread-state> ??</code></p> <p>where:</p> <p><code><thread-state></code> is one of the following: pending staged</p>	<p><code>locality#*/total or locality#*/worker-thread#*</code> or <code>locality#*/pool#*/worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <i>locality</i> for which the average wait time of HPX-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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> 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 HPX-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 <code>total</code> the counter returns the wait time of HPX-threads of the given state for all worker threads (cores) on that <i>locality</i>. If the instance name is <code>worker-thread#*</code> the counter will return the wait time of HPX-threads in the given state for all worker threads separately. These counters are available only if the compile time constant <code>HPX_WITH_THREAD_QUEUE</code> was defined while compiling the HPX core library (default: OFF). The unit of measure for this counter is nanosecond [ns].</p>	<p>None</p> <p><small>WAITTIME</small></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>	<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).</p>	None
-----------------------	--	--	------

continues on next page

Table 2.29 – continued from previous page

/threads/ creation-idle-rate ??	<p><code>locality#*/total</code> or <code>locality#*/</code> <code>worker-thread#*</code> or <code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <code>locality</code> for which the average creation idle rate of all (or one) worker threads should be queried for. The <code>locality</code> id (given by * is a (zero based) number identifying the <code>locality</code>.</p> <p><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> 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 <code>locality</code> 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><code>locality#*/total</code> or <code>locality#*/</code> <code>worker-thread#*</code> or <code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> 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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> 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 ??	locality#*/total or locality#*/ worker-thread#* or locality#*/pool#*/ worker-thread#* where: 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> . pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current length of all thread queues in the scheduler should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code> . If no pool-name is specified the counter refers to the ‘default’ pool.	Returns the overall length of all queues for the given worker thread(s) on the given <i>locality</i> .	None
/threads/count/ stack-unbinds ??	locality#*/total where: * 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> .	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 HPX-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 HPX-threads ‘stolen’ from the pending thread queue by a neighboring thread worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).	None

continues on next page

Table 2.29 – continued from previous page

/threads/count/pending-misses ??	<p><code>locality#*/total</code> or <code>locality#*/worker-thread#*</code> or <code>locality#/pool#/worker-thread#*</code></p> <p>where: <code>locality#*</code> 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>) <code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for. <code>worker-thread#*</code> is defining the worker thread for which the number of pending queue misses should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <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><code>locality#*/total</code> or <code>locality#*/worker-thread#*</code> or <code>locality#*/pool#*/worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> 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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> 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><code>locality#*/total</code> or <code>locality#*/</code> <code>worker-thread#*</code> or <code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <i>locality</i> for which the number of HPX-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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> is defining the worker thread for which the number of HPX-threads stolen from the staged queue should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	<p>Returns the total number of HPX-threads ‘stolen’ from the staged thread queue by a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).</p>	None
---	--	---	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/ stolen-to-pending ??	<p><code>locality#*/total</code> or <code>locality#*/</code> <code>worker-thread#*</code> or <code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <i>locality</i> for which the number of HPX-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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> is defining the worker thread for which the number of HPX-threads stolen to the pending queue should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the total number of HPX-threads ‘stolen’ to the pending thread queue of the worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant <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><code>locality#*/total</code> or <code>locality#*/</code> <code>worker-thread#*</code></p> <p>or</p> <p><code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> is defining the <i>locality</i> for which the number of HPX-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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> is defining the worker thread for which the number of HPX-threads stolen to the staged queue should be queried for. The worker thread number (given by the * is a (zero based) worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	Returns the total number of HPX-threads ‘stolen’ to the staged thread queue of a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).	None
---	---	--	------

continues on next page

Table 2.29 – continued from previous page

/threads/count/ objects ??	<code>locality#*/total</code> or <code>locality#*/ allocator#*</code> where: <code>locality#*</code> is defining the <i>locality</i> for which the current (cumulative) number 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> . <code>allocator#*</code> is defining the number of the allocator instance using which the threads have been created. <i>HPX</i> uses a varying number of allocators to create (and recycle) <i>HPX</i> -thread objects, most likely these counters are of use for debugging purposes only. The allocator id (given by * is a (zero based) number identifying the allocator to query.	Returns the total number of <i>HPX</i> -thread objects created. Note that thread objects are reused to improve system performance, thus this number does not reflect the number of actually executed (retired) <i>HPX</i> -threads.	None
/scheduler/ utilization/ instantaneous ??	<code>locality#*/total</code> where: <code>locality#*</code> 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 ??	<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 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>.</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 idle-loop counter should be queried for. The worker thread number (given by the * is a (zero based) worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the ‘default’ pool.</p>	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><code>locality#*/</code> <code>worker-thread#*</code></p> <p>or</p> <p><code>locality#*/pool#*/</code> <code>worker-thread#*</code></p> <p>where:</p> <p><code>locality#*</code> 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><code>pool#*</code> is defining the pool for which the current value of the idle-loop counter should be queried for.</p> <p><code>worker-thread#*</code> 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 or locality#*/ worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <i>--hpx:threads</i> .	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 num- ber (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 <i>--hpx:threads</i> .	Returns the background overhead on the given locality since applica- tion 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 con- figuration time constants <code>HPX_WITH_BACKGROUND_THREAD_COUNTERS</code> (default: OFF) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%.	None
--	---	--	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/background-send-duration??	locality#*/total or locality#*/ worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work related to sending parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work related to sending parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option -- <i>hpx:threads</i> .	Returns the overall time spent performing background work related to sending parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcelport only.	None
--	--	---	------

continues on next page

Table 2.29 – continued from previous page

/threads/ background-send-overhead ??	locality#*/total or locality#*/ worker-thread#* where: locality#* is defining the locality for which the background overhead re- lated to sending parcels should be queried for. The locality 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 sending parcels should be queried for. The worker thread num- ber (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 <i>--hpx:threads</i> .	Returns the background overhead related to sending 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 con- figuration time constants <code>HPX_WITH_BACKGROUND_THREAD_COUNTERS</code> (default: OFF) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcelport only.	None
---	---	--	------

continues on next page

Table 2.29 – continued from previous page

/threads/time/background-receive-duration??	locality#*/total or locality#*/ worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work related to receiving parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work related to receiving parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <code>--hpx:threads</code> .	Returns the overall time spent performing background work related to receiving parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcelport only.	None
---	---	---	------

continues on next page

Table 2.29 – continued from previous page

/threads/ background-receive-overhead??	locality#*/total or locality#*/ worker-thread#* where: locality#* is defining the locality for which the background overhead related to receiving should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the background overhead related to receiving parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option <i>--hpx:threads</i> .	Returns the background overhead related to receiving parcels on the given locality since application start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcelport only.	None
--	---	---	------

Table 2.30: 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 locality the number of action invocations should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall (local) invocation count of the specified action type on the given <i>locality</i> .	The action type. This is the string which has been used while registering the action with <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_ACTION</i> or <i>HPX_REGISTER_ACTION_ID</i> .
/runtime/count/remote-action-invocation ??	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the number of action invocations should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall (remote) invocation count of the specified action type on the given <i>locality</i> .	The action type. This is the string which has been used while registering the action with <i>HPX</i> , e.g. which has been passed as the second parameter to the macro <i>HPX_REGISTER_ACTION</i> or <i>HPX_REGISTER_ACTION_ID</i> .
/runtime/uptime ??	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the system uptime should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the overall time since application start on the given <i>locality</i> in nanoseconds.	None
/runtime/memory/virtual ??	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the allocated virtual memory should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the amount of virtual memory currently allocated by the referenced <i>locality</i> (in bytes).	None
/runtime/memory/resident ??	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> the allocated resident memory should be queried. The <i>locality</i> id is a (zero based) number identifying the <i>locality</i> .	Returns the amount of resident memory currently allocated by the referenced <i>locality</i> (in bytes).	None
2.3. Manual	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i> identifying the <i>locality</i> .		205
/runtime/memory/total ??	locality#*/total where: * is the <i>locality</i> id of the <i>locality</i>	Returns the total available <i>locality</i> (in bytes). This counter is	None memory for use by the referenced

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=all 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 using the option <i>--hpx:threads</i> .	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
/ stat? ??	Any full performance counter average name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name).	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume 1000 [ms] as the default. The second value can be either 0 or 1 and specifies whether the underlying counter should be reset during evaluation 1 or not 0. The default value is 0.
/ stat? ??	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.
/ stat? ??	Any full performance counter stddev name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current standard deviation (stddev) value calculated based on the values queried from the underlying counter (the one specified as the instance name).	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume 1000 [ms] as the default. The second value can be either 0 or 1 and specifies whether the underlying counter should be reset during evaluation 1 or not 0. The default value is 0.
/ stat? ??	Any full performance counter rolling stddev 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.
/ stat? ??	Any full performance counter median name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current (statistically estimated) median value calculated based on the values queried from the underlying counter (the one specified as the instance name).	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume 1000 [ms] as the default. The second value can be either 0 or 1 and specifies whether the underlying counter should be reset during evaluation 1 or not 0. The default value is 0.
2.3. Manual			

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.
208		Chapter 2. What's so special about HPX?	
/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	Description	Parameters
/ locality#*/ coalescing/ count/where: parcel's is 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 <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> .	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> .
/ locality#*/ coalescing/ count/where: message's is 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 <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> .	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> .
/ locality#*/ coalescing/ count/where: average-parcels-per-message? ??	Returns the average number of parcels sent in a message generated by the message handler associated with the action which is given by the <i>message</i> parameter. 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> .	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>
210 / locality#*/ coalescing/ time/ where: average-parcel-karrival	Returns the average time between arriving parcels for the action which is given by the counter parameter.	Chapter 2. What's so special about HPX? 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

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_ACTIONUSES_MESSAGE_COALESCING` and `HPX_ACTIONUSES_MESSAGE_COALESCING_NOTHROW`).

APEX integration

HPX provides integration with [APEX](#)¹³⁷, which is a framework for application profiling using task timers and various performance counters Huck *et al.*¹⁴⁰. It can be added as a git submodule by turning on the option `HPX_WITH_APEX:BOOL` during CMake configuration. [TAU](#)¹³⁸ 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](#)¹³⁹ for detailed instructions on using APEX with *HPX*.

References

2.3.13 *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)

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 using the FIFO (first-in-first-out) queing policy. This policy can be invoked using `--hpx:queuing=local-priority-fifo`. The scheduler can also be enabled using the LIFO (last-in-first-out) policy. This is not the default policy and must be invoked using the command line option `--hpx:queuing=local-priority-lifo`.

¹³⁷ <http://uo-oaciss.github.io/apex>

¹⁴⁰ 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.

¹³⁸ <https://www.cs.uoregon.edu/research/tau/home.php>

¹³⁹ <https://uo-oaciss.github.io/apex/usage/#hpx-louisiana-state-university>

Static priority scheduling policy

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

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

Local scheduling policy

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

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

Static scheduling policy

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

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

Priority ABP scheduling policy

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

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

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

The HPX resource partitioner

The *HPX* resource partitioner lets you take the execution resources available on a system—processing units, cores, and numa domains—and assign them to thread pools. By default *HPX* creates a single thread pool name `default`. While this is good for most use cases, the resource partitioner lets you create multiple thread pools with custom resources and options.

Creating custom thread pools is useful for cases where you have tasks which absolutely need to run without interference from other tasks. An example of this is when using `MPI`¹⁴¹ for distribution instead of the built-in mechanisms in

¹⁴¹ https://en.wikipedia.org/wiki/Message_Passing_Interface

HPX (useful in legacy applications). In this case one can create a thread pool containing a single thread for MPI communication. MPI 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 \(HWLOC\)](#)¹⁴² 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 are 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.

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();
}
```

¹⁴² <https://www.open-mpi.org/projects/hwloc/>

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");

    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())
    {
```

(continues on next page)

(continued from previous page)

```

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`.

2.3.14 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";
```

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

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

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

```

}
hpx::cout << std::flush;
//]

// 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";
}

```

Working with error codes

Most of the API functions exposed by *HPX* can be invoked in two different modes. By default those will throw an exception on error as described above. However, sometimes it is desirable not to throw an exception in case of an error condition. In this case an object instance of the `hpx::error_code` type can be passed as the last argument to the API function. In case of an error, the error condition will be returned in that `hpx::error_code` instance. The following example demonstrates extracting the full diagnostic information without exception handling:

```

hpx::cout << "{stack-trace}: " << hpx::get_error_backtrace(e)
    << "\n";
    hpx::cout << "{env}: " << hpx::get_error_env(e) << "\n";
}
hpx::cout << std::flush;
//]

///////////////////////////////
// Error reporting using error code
{
    //#[error_handling_diagnostic_information
    hpx::cout << "Error reporting using error code\n";

    // Create a new error_code instance.

```

(continues on next page)

(continued from previous page)

```

    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);

```

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:

```

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

    hpx::cout << std::flush;
//]
}

// 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";
    }
}

```

(continues on next page)

(continued from previous page)

<code>hpx::cout << "{function}: " << hpx::get_error_function_name(ec)</code>
--

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:

```

        hpx::cout << "{thread-id}: " << std::hex
                    << hpx::get_error_thread_id(ec) << "\n";
        hpx::cout << "{thread-description}: "
                    << hpx::get_error_thread_description(ec) << "\n\n";
        hpx::cout << "{state}: " << std::hex << hpx::get_error_state(ec)
                    << "\n";
        hpx::cout << "{stack-trace}: " << hpx::get_error_backtrace(ec)
                    << "\n";
        hpx::cout << "{env}: " << hpx::get_error_env(ec) << "\n";
    }

    hpx::cout << std::flush;
    //]
}

// 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.
}

```

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`.

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" << std::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(
```

(continues on next page)

(continued from previous page)

```
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.

2.3.15 Troubleshooting

Common issues

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

See also the closed issues on [GitHub¹⁴³](#) to find out how other people resolved a similar problem. If nothing of that works, you can also open a new issue on [GitHub¹⁴⁴](#) or contact us using one the options found in [Support for deploying and using HPX¹⁴⁵](#).

Undefined reference to `hpx::cout`

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

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

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.

Fail compiling for examples with `hpx::future` and `co_await`

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

```
error: coroutines require a traits template; cannot find 'std::coroutine_traits'
```

This can be resolved by using `-DHFX_WITH_CXX_STANDARD=20` to the `cmake` command line. Note that a compiler that supports C++20 is needed.

See also the corresponding closed Issue #5784¹⁴⁶.

¹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues?q=is%3Aissue+is%3Aclosed>

¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues>

¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/blob/master/.github/SUPPORT.md>

¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5784>

Build fails with ASIO error

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

```
Cannot open include file asio/io_context.hpp
```

This can be resolved by using `-DHPX_WITH_FETCH_ASIO=ON` to the cmake command line.

See also the corresponding closed Issue #5404¹⁴⁷ for more information.

Build fails with TCMalloc error

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

```
Could NOT find TCMalloc (missing: TCMALLOC_LIBRARY TCMALLOC_INCLUDE_DIR)
ERROR: HPX_WITH_MALLOC was set to tcmalloc, but tcmalloc could not be
found. Valid options for HPX_WITH_MALLOC are: system, tcmalloc, jemalloc,
mimalloc, tbbmalloc, and custom
```

This can be resolved either by defining `HPX_WITH_MALLOC=system` or by installing TCMalloc. This error occurs when users don't specify an option for `HPX_WITH_MALLOC`; in that case, `HPX` will be looking `tcmalloc`, which is the default value.

Useful suggestions

Reducing compilation time

If you want to significantly reduce compilation time, you can just use the local part of `HPX` for parallelism by disabling the distributed functionality. Moreover, you can avoid compiling examples. These can be done with the following flags:

```
-DHPX_WITH_NETWORKING=OFF
-DHPX_WITH_DISTRIBUTED_RUNTIME=OFF
-DHPX_WITH_EXAMPLES=OFF
-DHPX_WITH_TESTS=OFF
```

Linking `HPX` to your application

If you want to avoid installing and linking `HPX`, you can just build `HPX` and then use the following flag on your `HPX` application CMake configuration:

```
-DHPX_DIR=<build_dir>/lib/cmake/HPX
```

Note: For this to work you need not to specify `-DCMAKE_INSTALL_PREFIX` when building `HPX`.

¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5404>

HPX-application build type conformance

Your application's build type should align with the HPX build type. For example, if you specified `-DCMAKE_BUILD_TYPE=Debug` during the *HPX* compilation, then your application needs to be compiled with the same flag. We recommend keeping a separate build folder for different build types and just point accordingly to the type you want by using `-DHPX_DIR=<build_dir>/lib/cmake/HPX`.

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 PGAS¹⁴⁸ models. Partitioned Global Address Space is passive in their means of address translation. Copy semantics, distributed compound operations, and affinity relationships are some of the global functionality supported by AGAS.

Process The concept of the “process” in *HPX* is extended beyond that of either sequential execution or communicating sequential processes. While the notion of process suggests action (as do “function” or “subroutine”) it has a further responsibility of context, that is, the logical container of program state. It is this aspect of operation that process is employed in *HPX*. Furthermore, referring to “parallel processes” in *HPX* designates the presence of parallelism within the context of a given process, as well as the coarse grained parallelism achieved through concurrency of multiple processes of an executing user job. *HPX* processes provide a hierarchical name space within the framework of the active global address space and support multiple means of internal state access from external sources.

Parcel The Parcel is a component in *HPX* that communicates data, invokes an action at a distance, and distributes flow-control through the migration of continuations. Parcels bridge the gap of asynchrony between synchronous domains while maintaining symmetry of semantics between local and global execution. Parcels enable message-driven computation and may be seen as a form of “active messages”. Other important forms of message-driven computation predating active messages include dataflow tokens¹⁴⁹, the J-machine’s¹⁵⁰ support for remote method instantiation, and at the coarse grained variations of Unix remote procedure calls, among others. This enables work to be moved to the data as well as performing the more common action of bringing data to the work. A parcel can cause actions to occur remotely and asynchronously, among which are the creation of threads at different system nodes or synchronous domains.

Local Control Object

Lightweight Control Object

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

¹⁴⁸ <https://www.pgas.org/>

¹⁴⁹ http://en.wikipedia.org/wiki/Dataflow_architecture

¹⁵⁰ <http://en.wikipedia.org/wiki/J%20Machine>

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

See [Using LCOs](#) for more details on how to use LCOs in *HPX*.

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

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

2.5 Why *HPX*?

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

2.5.1 ParalleX—a new execution model for future architectures

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

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

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

2.5.2 What is HPX?

High Performance ParalleX (*HPX*) is the first runtime system implementation of the ParalleX execution model. The *HPX* runtime 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||AR¹⁵¹ Group (Systems Technology, Emergent Parallelism, and Algorithm Research) at Louisiana State University (LSU)¹⁵²'s Center for Computation and Technology (CCT)¹⁵³ to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STE||AR¹⁵⁴ is a research group focusing on system software solutions and scientific application development for hybrid and many-core hardware architectures.

For more information about the STE||AR¹⁵⁵ 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 FLOPS¹⁵⁶—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 Law¹⁵⁷) and weak scaling (see Gustafson's Law¹⁵⁸). 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

¹⁵¹ <https://stellar-group.org>

¹⁵² <https://www.lsu.edu>

¹⁵³ <https://www.cct.lsu.edu>

¹⁵⁴ <https://stellar-group.org>

¹⁵⁵ <https://stellar-group.org>

¹⁵⁶ <http://en.wikipedia.org/wiki/FLOPS>

¹⁵⁷ http://en.wikipedia.org/wiki/Amdahl%27s_law

¹⁵⁸ http://en.wikipedia.org/wiki/Gustafson%27s_law

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 Neumann¹⁵⁹, as published back in 1945, and later extended by the Harvard architecture¹⁶⁰. 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:

- Performance: as previously mentioned, scalability and efficiency are the main criteria people are interested in.
- Fault tolerance: the low expected mean time between failures (MTBF¹⁶¹) 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.

¹⁵⁹ <http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf>

¹⁶⁰ http://en.wikipedia.org/wiki/Harvard_architecture

¹⁶¹ http://en.wikipedia.org/wiki/Mean_time_between_failures

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 InfiniBand¹⁶², caching memory hierarchies in all modern processors, the constant optimization of existing MPI¹⁶³ implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use GPGPUs¹⁶⁴ 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.

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 XMT¹⁶⁵). 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)¹⁶⁶, Microsoft Parallel Patterns Library (PPL)¹⁶⁷, Cilk++¹⁶⁸, 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 OpenMP¹⁶⁹) or processes (MPI¹⁷⁰). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP¹⁷¹ as the system synchronizes the threads used to execute the different iterations in parallel. In MPI¹⁷² 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

¹⁶² <http://en.wikipedia.org/wiki/InfiniBand>

¹⁶³ https://en.wikipedia.org/wiki/Message_Passing_Interface

¹⁶⁴ <http://en.wikipedia.org/wiki/GPGPU>

¹⁶⁵ http://en.wikipedia.org/wiki/Cray_XMT

¹⁶⁶ <https://www.threadingbuildingblocks.org/>

¹⁶⁷ <https://msdn.microsoft.com/en-us/library/dd492418.aspx>

¹⁶⁸ <https://software.intel.com/en-us/articles/intel-cilk-plus/>

¹⁶⁹ <https://openmp.org/wp/>

¹⁷⁰ https://en.wikipedia.org/wiki/Message_Passing_Interface

¹⁷¹ <https://openmp.org/wp/>

¹⁷² https://en.wikipedia.org/wiki/Message_Passing_Interface

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.

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 PGAS¹⁷³ (Partitioned Global Address Space) designed to overcome this limitation, such as Chapel¹⁷⁴, X10¹⁷⁵, UPC¹⁷⁶, or Fortress¹⁷⁷. 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++¹⁷⁸. The first attempts towards solving related problem go back decades as well, a good example is the Linda coordination language¹⁷⁹. 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 GPGPUs¹⁸⁰ 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,

¹⁷³ <https://www.pgas.org/>

¹⁷⁴ <https://chapel.cray.com/>

¹⁷⁵ <https://x10-lang.org/>

¹⁷⁶ <https://upc.lbl.gov/>

¹⁷⁷ <https://labs.oracle.com/projects/plrg/Publications/index.html>

¹⁷⁸ <https://charm.cs.uiuc.edu/>

¹⁷⁹ [http://en.wikipedia.org/wiki/Linda_\(coordination_language\)](http://en.wikipedia.org/wiki/Linda_(coordination_language))

¹⁸⁰ <http://en.wikipedia.org/wiki/GPGPU>

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++¹⁸¹), we see better asynchrony, simpler application codes, and improved scaling.

Favor message driven computation over message passing

Today's prevalently used programming model on parallel (multi-node) systems is MPI. It is based on message passing, as the name implies, which means that the receiver has to be aware of a message about to come in. Both codes, the sender and the receiver, have to synchronize in order to perform the communication step. Even the newer, asynchronous interfaces require explicitly coding the algorithms around the required communication scheme. As a result, everything but the most trivial MPI applications spends a considerable amount of time waiting for incoming messages, thus, causing starvation and latencies to impede full resource utilization. The more complex and more dynamic the data structures and algorithms become, the larger the adverse effects. The community discovered message-driven and data-driven methods of implementing algorithms a long time ago, and systems such as Charm++¹⁸² 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 lectures¹⁸³
 - Slides¹⁸⁴
- Tutorials repository¹⁸⁵
- STE||AR Group blog posts¹⁸⁶
- Basic *HPX* recipes
 - Exporting a free function from a shared library which lives in a namespace, to use as Action¹⁸⁷
 - Turning a struct or class into a component and use its methods¹⁸⁸
 - Creating and referencing components in hpx¹⁸⁹

¹⁸¹ <https://charm.cs.uiuc.edu/>

¹⁸² <https://charm.cs.uiuc.edu/>

¹⁸³ <https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOnIJ0lFNjKze18>

¹⁸⁴ <https://github.com/STELLAR-GROUP/tutorials/tree/master/cscs2016>

¹⁸⁵ <https://github.com/STELLAR-GROUP/tutorials>

¹⁸⁶ <http://stellar-group.org/blog/>

¹⁸⁷ <https://gitlab.com/-/snippets/1821389>

¹⁸⁸ <https://gitlab.com/-/snippets/1822983>

¹⁸⁹ <https://gitlab.com/-/snippets/1828131>

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.

Note: There is a dependency report that displays useful information about the structure of the code. It is available for each commit at [HPX Dependency report](#).

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. Those wait for all futures to become ready before returning to the user. Note that the `wait_*` functions will rethrow one of the exceptions from exceptional futures. The `wait_*_nothrow` combinators are equivalent to the `wait_*` functions exception that they do not throw if one of the futures has become exceptional.

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&&)
{
    // 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)
    });
}
);

```

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

(continues on next page)

(continued from previous page)

```
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.

compute_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::detail::small_vector`
- `hpx::util::basic_any`
- `hpx::util::member_pack`
- `hpx::optional`
- `hpx::util::tuple`
- `hpx::variant`

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 P0443¹⁹⁰ 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.

¹⁹⁰ <http://wg21.link/p0443>

- **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.

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::function`
- `hpx::function_ref_`
- `hpx::move_only_function`
- `hpx::bind`
- `hpx::bind_back`
- `hpx::bind_front`

- `hpx::util::deferred_call`
- `hpx::invoke`
- `hpx::invoke_r`
- `hpx::invoke_fused`
- `hpx::invoke_fused_r`
- `hpx::mem_fn`
- `hpx::util::one_shot`
- `hpx::util::protect`
- `hpx::util::result_of`
- `hpx::placeholders::_1`
- `hpx::placeholders::_2`
- ...
- `hpx::placeholders::_9`

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 VTune¹⁹¹.

See the API reference of this module for more details.

¹⁹¹ <https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html>

lci_base

This module provides helper functionality for detecting MPI environments.

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::packaged_task`
- `hpx::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.

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 [here¹⁹²](#). 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 [P2220¹⁹³](#). This differs from [P1393¹⁹⁴](#) in that it relies fully on `tag_invoke` overloads and fewer base customization points. Actual properties are defined in modules. All functionality is experimental and can be accessed through the `hpx::experimental` namespace.

See the API reference of this module for more details.

resiliency

In *HPX*, a program failure is a manifestation of a failing task. This module exposes several APIs that allow users to manage failing tasks in a convenient way by either replaying a failed task or by replicating a specific task.

Task replay is analogous to the Checkpoint/Restart mechanism found in conventional execution models. The key difference being localized fault detection. When the runtime detects an error, it replays the failing task as opposed to completely rolling back the entire program to the previous checkpoint.

Task replication is designed to provide reliability enhancements by replicating a set of tasks and evaluating their results to determine a consensus among them. This technique is most effective in situations where there are few tasks in the critical path of the DAG which leaves the system underutilized or where hardware or software failures may result in an incorrect result instead of an error. However, the drawback of this method is the additional computational cost incurred by repeating a task multiple times.

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.

¹⁹² https://www.boost.org/doc/libs/1_70_0/doc/html/program_options.html

¹⁹³ <https://wg21.link/p2220>

¹⁹⁴ <http://wg21.link/p1393>

- `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 model¹⁹⁵ 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.

¹⁹⁵ 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 that should be used rather than the C++ standard ones in *HPX* threads:

- `hpx::barrier`
- `hpx::binary_semaphore`
- `hpx::call_once`
- `hpx::condition_variable`
- `hpx::condition_variable_any`
- `hpx::counting_semaphore`
- `hpx::lcos::local::event`
- `hpx::latch`
- `hpx::mutex`
- `hpx::no_mutex`
- `hpx::once_flag`
- `hpx::recursive_mutex`
- `hpx::shared_mutex`
- `hpx::sliding_semaphore`
- `hpx::spinlock` (`std::mutex` compatible spinlock)
- `hpx::spinlock_no_backoff` (`boost::mutex` compatible spinlock)
- `hpx::spinlock_pool`
- `hpx::stop_callback`
- `hpx::stop_source`
- `hpx::stop_token`
- `hpx::in_place_stop_token`
- `hpx::timed_mutex`
- `hpx::upgrade_to_unique_lock`
- `hpx::upgrade_lock`

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)¹⁹⁶ 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.

¹⁹⁶ <https://www.open-mpi.org/projects/hwloc/>

agas

TODO: High-level description of the module.

See the [API reference](#) of this module for more details.

agas_base

This module holds the implementation of the four AGAS services: primary namespace, locality namespace, component namespace, and symbol namespace.

See the [API reference](#) of this module for more details.

async_colocated

TODO: High-level description of the module.

See the [API reference](#) of this module for more details.

async

This module contains functionality for asynchronously launching work on remote localities: `hpx::async`, `hpx::apply`. This module extends the local-only functions in `libs_async_local`.

See the API reference of this module for more details.

checkpoint

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

Found in `hpx/util/checkpoint.hpp`, `checkpoints` are defined as objects that hold a serialized version of an object or set of objects at a particular moment in time. This representation can be stored in memory for later use or it can be written to disk for storage and/or recovery at a later point. In order to create and fill this object with data, users must use a function called `save_checkpoint`. In code the function looks like this:

```
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);
```

(continues on next page)

(continued from previous page)

```

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 ostream
          archive7.end(),   // ie. the file
          std::ostream_iterator<char>(test_file_7));
test_file_7.close();

std::vector<float> vec7_1;
std::vector<char> char_vec;
std::ifstream test_file_7_1("checkpoint_test_file.txt");
if (test_file_7_1)
{
    test_file_7_1.seekg(0, test_file_7_1.end);
    auto length = test_file_7_1.tellg();
    test_file_7_1.seekg(0, test_file_7_1.beg);
    char_vec.resize(length);
    test_file_7_1.read(char_vec.data(), length);
}
checkpoint archive7_1(std::move(char_vec));    // Write data to checkpoint
restore_checkpoint(archive7_1, vec7_1);

```

Checkpointing components

`save_checkpoint` and `restore_checkpoint` are also able to store components inside checkpoints. This can be done in one of two ways. First a client of the component can be passed to `save_checkpoint`. When the user wishes to resurrect the component she can pass a client instance to `restore_checkpoint`.

This technique is demonstrated below:

```
// Try to checkpoint and restore a component with a client
std::vector<int> vec3{10, 10, 10, 10, 10};

// Create a component instance through client constructor
data_client D(hpx::find_here(), std::move(vec3));
hpx::future<checkpoint> f3 = save_checkpoint(D);

// Create a new client
data_client E;

// Restore server inside client instance
restore_checkpoint(f3.get(), E);
```

The second way a user can save a component is by passing a `shared_ptr` to the component to `save_checkpoint`. This component can be resurrected by creating a new instance of the component type and passing a `shared_ptr` to the new instance to `restore_checkpoint`.

This technique is demonstrated below:

```
// test checkpoint a component using a shared_ptr
std::vector<int> vec{1, 2, 3, 4, 5};
data_client A(hpx::find_here(), std::move(vec));

// Checkpoint Server
hpx::id_type old_id = A.get_id();

hpx::future<std::shared_ptr<data_server>> f_a_ptr =
    hpx::get_ptr<data_server>(A.get_id());
std::shared_ptr<data_server> a_ptr = f_a_ptr.get();
hpx::future<checkpoint> f = save_checkpoint(a_ptr);
auto&& data = f.get();

// test prepare_checkpoint API
checkpoint c = prepare_checkpoint(hpx::launch::sync, a_ptr);
HPX_TEST(c.size() == data.size());

// Restore Server
// Create a new server instance
std::shared_ptr<data_server> b_server;
restore_checkpoint(data, b_server);
```

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::distributed::barrier`: distributed barrier.
- `hpx::lcos::fold`: performs a fold with a given action on all given global identifiers.
- `hpx::distributed::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_lci

TODO: High-level description of the module.

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

Our API is semantically conforming; hence, the reader is highly encouraged to refer to the corresponding facility in the [C++ Standard](#)¹⁹⁷ if needed. 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.

¹⁹⁷ <https://en.cppreference.com/w/cpp/header>

hpx/algorithm.hpp

The header `hpx/algorithm.hpp`¹⁹⁸ corresponds to the C++ standard library header `algorithm`¹⁹⁹. See [Using parallel algorithms](#) for more information about the parallel algorithms.

ClassesTable 2.35: Classes of header `hpx/algorithm.hpp`

Class	C++ standard
<code>hpx::experimental::reduction</code>	N4808 ²⁰⁰
<code>hpx::experimental::induction</code>	N4808 ²⁰¹

FunctionsTable 2.36: `hpx` functions of header `hpx/algorithm.hpp`

<code>hpx</code> function	C++ standard
<code>hpx::adjacent_difference</code>	<code>std::adjacent_difference</code> ²⁰²
<code>hpx::adjacent_find</code>	<code>std::adjacent_find</code> ²⁰³
<code>hpx::all_of</code>	<code>std::all_of</code> ²⁰⁴
<code>hpx::any_of</code>	<code>std::any_of</code> ²⁰⁵
<code>hpx::copy</code>	<code>std::copy</code> ²⁰⁶
<code>hpx::copy_if</code>	<code>std::copy_if</code> ²⁰⁷
<code>hpx::copy_n</code>	<code>std::copy_n</code> ²⁰⁸
<code>hpx::count</code>	<code>std::count</code> ²⁰⁹
<code>hpx::count_if</code>	<code>std::count_if</code> ²¹⁰
<code>hpx::ends_with</code>	<code>std::ends_with</code> ²¹¹
<code>hpx::equal</code>	<code>std::equal</code> ²¹²
<code>hpx::fill</code>	<code>std::fill</code> ²¹³
<code>hpx::fill_n</code>	<code>std::fill_n</code> ²¹⁴
<code>hpx::find</code>	<code>std::find</code> ²¹⁵
<code>hpx::find_end</code>	<code>std::find_end</code> ²¹⁶
<code>hpx::find_first_of</code>	<code>std::find_first_of</code> ²¹⁷
<code>hpx::find_if</code>	<code>std::find_if</code> ²¹⁸
<code>hpx::find_if_not</code>	<code>std::find_if_not</code> ²¹⁹
<code>hpx::for_each</code>	<code>std::for_each</code> ²²⁰
<code>hpx::for_each_n</code>	<code>std::for_each_n</code> ²²¹
<code>hpx::generate</code>	<code>std::generate</code> ²²²
<code>hpx::generate_n</code>	<code>std::generate_n</code> ²²³
<code>hpx::includes</code>	<code>std::includes</code> ²²⁴
<code>hpx::inplace_merge</code>	<code>std::inplace_merge</code> ²²⁵
<code>hpx::is_heap</code>	<code>std::is_heap</code> ²²⁶
<code>hpx::is_heap_until</code>	<code>std::is_heap_until</code> ²²⁷
<code>hpx::is_partitioned</code>	<code>std::is_partitioned</code> ²²⁸

continues on next page

¹⁹⁸ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/algorithm.hpp>

¹⁹⁹ <http://en.cppreference.com/w/cpp/header/algorithm>

²⁰⁰ <http://wg21.link/n4808>

²⁰¹ <http://wg21.link/n4808>

Table 2.36 – continued from previous page

<i>hpx</i> function	C++ standard
<i>hpx::is_sorted</i>	std::is_sorted ²²⁹
<i>hpx::is_sorted_until</i>	std::is_sorted_until ²³⁰
<i>hpx::lexicographical_compare</i>	std::lexicographical_compare ²³¹
<i>hpx::make_heap</i>	std::make_heap ²³²
<i>hpx::max_element</i>	std::max_element ²³³
<i>hpx::merge</i>	std::merge ²³⁴
<i>hpx::min_element</i>	std::min_element ²³⁵
<i>hpx::minmax_element</i>	std::minmax_element ²³⁶
<i>hpx::mismatch</i>	std::mismatch ²³⁷
<i>hpx::move</i>	std::move ²³⁸
<i>hpx::none_of</i>	std::none_of ²³⁹
<i>hpx::nth_element</i>	std::nth_element ²⁴⁰
<i>hpx::partial_sort</i>	std::partial_sort ²⁴¹
<i>hpx::partial_sort_copy</i>	std::partial_sort_copy ²⁴²
<i>hpx::partition</i>	std::partition ²⁴³
<i>hpx::partition_copy</i>	std::partition_copy ²⁴⁴
<i>hpx::parallel::v1::reduce_by_key</i>	reduce_by_key ²⁴⁵
<i>hpx::remove</i>	std::remove ²⁴⁶
<i>hpx::remove_copy</i>	std::remove_copy ²⁴⁷
<i>hpx::remove_copy_if</i>	std::remove_copy_if ²⁴⁸
<i>hpx::remove_if</i>	std::remove_if ²⁴⁹
<i>hpx::replace</i>	std::replace ²⁵⁰
<i>hpx::replace_copy</i>	std::replace_copy ²⁵¹
<i>hpx::replace_copy_if</i>	std::replace_copy_if ²⁵²
<i>hpx::replace_if</i>	std::replace_if ²⁵³
<i>hpx::reverse</i>	std::reverse ²⁵⁴
<i>hpx::reverse_copy</i>	std::reverse_copy ²⁵⁵
<i>hpx::rotate</i>	std::rotate ²⁵⁶
<i>hpx::rotate_copy</i>	std::rotate_copy ²⁵⁷
<i>hpx::search</i>	std::search ²⁵⁸
<i>hpx::search_n</i>	std::search_n ²⁵⁹
<i>hpx::set_difference</i>	std::set_difference ²⁶⁰
<i>hpx::set_intersection</i>	std::set_intersection ²⁶¹
<i>hpx::set_symmetric_difference</i>	std::set_symmetric_difference ²⁶²
<i>hpx::set_union</i>	std::set_union ²⁶³
<i>hpx::shift_left</i>	std::shift_left ²⁶⁴
<i>hpx::shift_right</i>	std::shift_right ²⁶⁵
<i>hpx::sort</i>	std::sort ²⁶⁶
<i>hpx::parallel::v1::sort_by_key</i>	sort_by_key ²⁶⁷
<i>hpx::stable_partition</i>	std::stable_partition ²⁶⁸
<i>hpx::stable_sort</i>	std::stable_sort ²⁶⁹
<i>hpx::starts_with</i>	std::starts_with ²⁷⁰
<i>hpx::swap_ranges</i>	std::swap_ranges ²⁷¹
<i>hpx::transform</i>	std::transform ²⁷²
<i>hpx::unique</i>	std::unique ²⁷³
<i>hpx::unique_copy</i>	std::unique_copy ²⁷⁴
<i>hpx::experimental::for_loop</i>	N4808 ²⁷⁵
<i>hpx::experimental::for_loop_strided</i>	N4808 ²⁷⁶
<i>hpx::experimental::for_loop_n</i>	N4808 ²⁷⁷

continues on next page

Table 2.36 – continued from previous page

<i>hpx</i> function	C++ standard
<code>hpx::experimental::for_loop_n_strided</code>	N4808 ²⁷⁸

202 http://en.cppreference.com/w/cpp/algorithm/adjacent_difference
203 http://en.cppreference.com/w/cpp/algorithm/adjacent_find
204 http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
205 http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
206 <http://en.cppreference.com/w/cpp/algorithm/copy>
207 <http://en.cppreference.com/w/cpp/algorithm/copy>
208 http://en.cppreference.com/w/cpp/algorithm/copy_n
209 <http://en.cppreference.com/w/cpp/algorithm/count>
210 <http://en.cppreference.com/w/cpp/algorithm/count>
211 http://en.cppreference.com/w/cpp/algorithm/ranges/ends_with
212 <http://en.cppreference.com/w/cpp/algorithm/equal>
213 <http://en.cppreference.com/w/cpp/algorithm/fill>
214 http://en.cppreference.com/w/cpp/algorithm/fill_n
215 <http://en.cppreference.com/w/cpp/algorithm/find>
216 http://en.cppreference.com/w/cpp/algorithm/find_end
217 http://en.cppreference.com/w/cpp/algorithm/find_first_of
218 <http://en.cppreference.com/w/cpp/algorithm/find>
219 <http://en.cppreference.com/w/cpp/algorithm/find>
220 http://en.cppreference.com/w/cpp/algorithm/for_each
221 http://en.cppreference.com/w/cpp/algorithm/for_each_n
222 <http://en.cppreference.com/w/cpp/algorithm/generate>
223 http://en.cppreference.com/w/cpp/algorithm/generate_n
224 <http://en.cppreference.com/w/cpp/algorithm/includes>
225 http://en.cppreference.com/w/cpp/algorithm/inplace_merge
226 http://en.cppreference.com/w/cpp/algorithm/is_heap
227 http://en.cppreference.com/w/cpp/algorithm/is_heap_until
228 http://en.cppreference.com/w/cpp/algorithm/is_partitioned
229 http://en.cppreference.com/w/cpp/algorithm/is_sorted
230 http://en.cppreference.com/w/cpp/algorithm/is_sorted_until
231 http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
232 http://en.cppreference.com/w/cpp/algorithm/make_heap
233 http://en.cppreference.com/w/cpp/algorithm/max_element
234 <http://en.cppreference.com/w/cpp/algorithm/merge>
235 http://en.cppreference.com/w/cpp/algorithm/min_element
236 http://en.cppreference.com/w/cpp/algorithm/minmax_element
237 <http://en.cppreference.com/w/cpp/algorithm/mismatch>
238 <http://en.cppreference.com/w/cpp/algorithm/move>
239 http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
240 http://en.cppreference.com/w/cpp/algorithm/nth_element
241 http://en.cppreference.com/w/cpp/algorithm/partial_sort
242 http://en.cppreference.com/w/cpp/algorithm/partial_sort_copy
243 <http://en.cppreference.com/w/cpp/algorithm/partition>
244 http://en.cppreference.com/w/cpp/algorithm/partition_copy
245 https://thrust.github.io/doc/group__reductions_gad5623f203f9b3fdcab72481c3913f0e0.html
246 <http://en.cppreference.com/w/cpp/algorithm/remove>
247 http://en.cppreference.com/w/cpp/algorithm/remove_copy
248 http://en.cppreference.com/w/cpp/algorithm/remove_copy
249 <http://en.cppreference.com/w/cpp/algorithm/remove>
250 <http://en.cppreference.com/w/cpp/algorithm/replace>
251 http://en.cppreference.com/w/cpp/algorithm/replace_copy
252 http://en.cppreference.com/w/cpp/algorithm/replace_copy
253 <http://en.cppreference.com/w/cpp/algorithm/replace>
254 <http://en.cppreference.com/w/cpp/algorithm/reverse>
255 http://en.cppreference.com/w/cpp/algorithm/reverse_copy
256 <http://en.cppreference.com/w/cpp/algorithm/rotate>
257 http://en.cppreference.com/w/cpp/algorithm/rotate_copy
258 <http://en.cppreference.com/w/cpp/algorithm/search>
259 http://en.cppreference.com/w/cpp/algorithm/search_n
260 http://en.cppreference.com/w/cpp/algorithm/set_difference
261 http://en.cppreference.com/w/cpp/algorithm/set_intersection
262 http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
263 http://en.cppreference.com/w/cpp/algorithm/set_union
264 <http://en.cppreference.com/w/cpp/algorithm/shift>
265 <http://en.cppreference.com/w/cpp/algorithm/shift>
266 <http://en.cppreference.com/w/cpp/algorithm/sort>
267 https://thrust.github.io/doc/group__sorting_gabe038d6107f7c824cf74120500ef45ea.html
268 http://en.cppreference.com/w/cpp/algorithm/stable_partition
269 http://en.cppreference.com/w/cpp/algorithm/stable_sort
270 http://en.cppreference.com/w/cpp/algorithm/ranges/starts_with
271 http://en.cppreference.com/w/cpp/algorithm/swap_ranges
272 <http://en.cppreference.com/w/cpp/algorithm/transform>
273 <http://en.cppreference.com/w/cpp/algorithm/unique>
274 http://en.cppreference.com/w/cpp/algorithm/unique_copy
275 <http://wo21.link/n4808>

Table 2.37: *hpx::ranges* functions of header `hpx/algorithm.hpp`

<i>hpx::ranges</i> function	C++ standard
<code>hpx::ranges::adjacent_find</code>	<code>std::adjacent_find</code> ²⁷⁹
<code>hpx::ranges::all_of</code>	<code>std::all_of</code> ²⁸⁰
<code>hpx::ranges::any_of</code>	<code>std::any_of</code> ²⁸¹
<code>hpx::ranges::copy</code>	<code>std::copy</code> ²⁸²
<code>hpx::ranges::copy_if</code>	<code>std::copy_if</code> ²⁸³
<code>hpx::ranges::copy_n</code>	<code>std::copy_n</code> ²⁸⁴
<code>hpx::ranges::count</code>	<code>std::count</code> ²⁸⁵
<code>hpx::ranges::count_if</code>	<code>std::count_if</code> ²⁸⁶
<code>hpx::ranges::ends_with</code>	<code>std::ends_with</code> ²⁸⁷
<code>hpx::ranges::equal</code>	<code>std::equal</code> ²⁸⁸
<code>hpx::ranges::fill</code>	<code>std::fill</code> ²⁸⁹
<code>hpx::ranges::fill_n</code>	<code>std::fill_n</code> ²⁹⁰
<code>hpx::ranges::find</code>	<code>std::find</code> ²⁹¹
<code>hpx::ranges::find_end</code>	<code>std::find_end</code> ²⁹²
<code>hpx::ranges::find_first_of</code>	<code>std::find_first_of</code> ²⁹³
<code>hpx::ranges::find_if</code>	<code>std::find_if</code> ²⁹⁴
<code>hpx::ranges::find_if_not</code>	<code>std::find_if_not</code> ²⁹⁵
<code>hpx::ranges::for_each</code>	<code>std::for_each</code> ²⁹⁶
<code>hpx::ranges::for_each_n</code>	<code>std::for_each_n</code> ²⁹⁷
<code>hpx::ranges::generate</code>	<code>std::generate</code> ²⁹⁸
<code>hpx::ranges::generate_n</code>	<code>std::generate_n</code> ²⁹⁹
<code>hpx::ranges::includes</code>	<code>std::includes</code> ³⁰⁰
<code>hpx::ranges::inplace_merge</code>	<code>std::inplace_merge</code> ³⁰¹
<code>hpx::ranges::is_heap</code>	<code>std::is_heap</code> ³⁰²
<code>hpx::ranges::is_heap_until</code>	<code>std::is_heap_until</code> ³⁰³
<code>hpx::ranges::is_partitioned</code>	<code>std::is_partitioned</code> ³⁰⁴
<code>hpx::ranges::is_sorted</code>	<code>std::is_sorted</code> ³⁰⁵
<code>hpx::ranges::is_sorted_until</code>	<code>std::is_sorted_until</code> ³⁰⁶
<code>hpx::ranges::make_heap</code>	<code>std::make_heap</code> ³⁰⁷
<code>hpx::ranges::merge</code>	<code>std::merge</code> ³⁰⁸
<code>hpx::ranges::move</code>	<code>std::move</code> ³⁰⁹
<code>hpx::ranges::none_of</code>	<code>std::none_of</code> ³¹⁰
<code>hpx::ranges::nth_element</code>	<code>std::nth_element</code> ³¹¹
<code>hpx::ranges::partial_sort</code>	<code>std::partial_sort</code> ³¹²
<code>hpx::ranges::partial_sort_copy</code>	<code>std::partial_sort_copy</code> ³¹³
<code>hpx::ranges::partition</code>	<code>std::partition</code> ³¹⁴
<code>hpx::ranges::partition_copy</code>	<code>std::partition_copy</code> ³¹⁵
<code>hpx::ranges::set_difference</code>	<code>std::set_difference</code> ³¹⁶
<code>hpx::ranges::set_intersection</code>	<code>std::set_intersection</code> ³¹⁷
<code>hpx::ranges::set_symmetric_difference</code>	<code>std::set_symmetric_difference</code> ³¹⁸
<code>hpx::ranges::set_union</code>	<code>std::set_union</code> ³¹⁹
<code>hpx::ranges::shift_left</code>	P2440 ³²⁰
<code>hpx::ranges::shift_right</code>	P2440 ³²¹
<code>hpx::ranges::sort</code>	<code>std::sort</code> ³²²
<code>hpx::ranges::stable_partition</code>	<code>std::stable_partition</code> ³²³
<code>hpx::ranges::stable_sort</code>	<code>std::stable_sort</code> ³²⁴
<code>hpx::ranges::starts_with</code>	<code>std::starts_with</code> ³²⁵
<code>hpx::ranges::swap_ranges</code>	<code>std::swap_ranges</code> ³²⁶

continues on next page

Table 2.37 – continued from previous page

<i>hpx::ranges</i> function	C++ standard
<i>hpx::ranges::unique</i>	<i>std::unique</i> ³²⁷
<i>hpx::ranges::unique_copy</i>	<i>std::unique_copy</i> ³²⁸
<i>hpx::ranges::experimental::for_loop</i>	N4808 ³²⁹
<i>hpx::ranges::experimental::for_loop_strided</i>	N4808 ³³⁰

- ²⁷⁹ http://en.cppreference.com/w/cpp/algorithm/ranges/adjacent_find
²⁸⁰ http://en.cppreference.com/w/cpp/algorithm/ranges/all_any_none_of
²⁸¹ http://en.cppreference.com/w/cpp/algorithm/ranges/all_any_none_of
²⁸² <http://en.cppreference.com/w/cpp/algorithm/ranges/copy>
²⁸³ <http://en.cppreference.com/w/cpp/algorithm/ranges/copy>
²⁸⁴ http://en.cppreference.com/w/cpp/algorithm/ranges/copy_n
²⁸⁵ <http://en.cppreference.com/w/cpp/algorithm/ranges/count>
²⁸⁶ <http://en.cppreference.com/w/cpp/algorithm/ranges/count>
²⁸⁷ http://en.cppreference.com/w/cpp/algorithm/ranges/ends_with
²⁸⁸ <http://en.cppreference.com/w/cpp/algorithm/ranges/equal>
²⁸⁹ <http://en.cppreference.com/w/cpp/algorithm/ranges/fill>
²⁹⁰ http://en.cppreference.com/w/cpp/algorithm/ranges/fill_n
²⁹¹ <http://en.cppreference.com/w/cpp/algorithm/ranges/find>
²⁹² http://en.cppreference.com/w/cpp/algorithm/ranges/find_end
²⁹³ http://en.cppreference.com/w/cpp/algorithm/ranges/find_first_of
²⁹⁴ <http://en.cppreference.com/w/cpp/algorithm/ranges/find>
²⁹⁵ <http://en.cppreference.com/w/cpp/algorithm/ranges/find>
²⁹⁶ http://en.cppreference.com/w/cpp/algorithm/ranges/for_each
²⁹⁷ http://en.cppreference.com/w/cpp/algorithm/ranges/for_each_n
²⁹⁸ <http://en.cppreference.com/w/cpp/algorithm/ranges/generate>
²⁹⁹ http://en.cppreference.com/w/cpp/algorithm/ranges/generate_n
³⁰⁰ <http://en.cppreference.com/w/cpp/algorithm/ranges/includes>
³⁰¹ http://en.cppreference.com/w/cpp/algorithm/ranges/inplace_merge
³⁰² http://en.cppreference.com/w/cpp/algorithm/ranges/is_heap
³⁰³ http://en.cppreference.com/w/cpp/algorithm/ranges/is_heap_until
³⁰⁴ http://en.cppreference.com/w/cpp/algorithm/ranges/is_partitioned
³⁰⁵ http://en.cppreference.com/w/cpp/algorithm/ranges/is_sorted
³⁰⁶ http://en.cppreference.com/w/cpp/algorithm/ranges/is_sorted_until
³⁰⁷ http://en.cppreference.com/w/cpp/algorithm/ranges/make_heap
³⁰⁸ <http://en.cppreference.com/w/cpp/algorithm/ranges/merge>
³⁰⁹ <http://en.cppreference.com/w/cpp/algorithm/ranges/move>
³¹⁰ http://en.cppreference.com/w/cpp/algorithm/ranges/all_any_none_of
³¹¹ http://en.cppreference.com/w/cpp/algorithm/ranges/nth_element
³¹² http://en.cppreference.com/w/cpp/algorithm/ranges/partial_sort
³¹³ http://en.cppreference.com/w/cpp/algorithm/ranges/partial_sort_copy
³¹⁴ <http://en.cppreference.com/w/cpp/algorithm/ranges/partition>
³¹⁵ http://en.cppreference.com/w/cpp/algorithm/ranges/partition_copy
³¹⁶ http://en.cppreference.com/w/cpp/algorithm/ranges/set_difference
³¹⁷ http://en.cppreference.com/w/cpp/algorithm/ranges/set_intersection
³¹⁸ http://en.cppreference.com/w/cpp/algorithm/ranges/set_symmetric_difference
³¹⁹ http://en.cppreference.com/w/cpp/algorithm/ranges/set_union
³²⁰ <https://wg21.link/p2440>
³²¹ <https://wg21.link/p2440>
³²² <http://en.cppreference.com/w/cpp/algorithm/ranges/sort>
³²³ http://en.cppreference.com/w/cpp/algorithm/ranges/stable_partition
³²⁴ http://en.cppreference.com/w/cpp/algorithm/ranges/stable_sort
³²⁵ http://en.cppreference.com/w/cpp/algorithm/ranges/starts_with
³²⁶ http://en.cppreference.com/w/cpp/algorithm/ranges/swap_ranges
³²⁷ <http://en.cppreference.com/w/cpp/algorithm/ranges/unique>
³²⁸ http://en.cppreference.com/w/cpp/algorithm/ranges/unique_copy
³²⁹ <http://wg21.link/n4808>
³³⁰ <http://wg21.link/n4808>

hpx/any.hpp

The header `hpx/any.hpp`³³¹ corresponds to the C++ standard library header `any`³³².

`hpx::any` is compatible with `std::any`.

Classes

Table 2.38: Classes of header `hpx/any.hpp`

Class	C++ standard
<code>hpx::any</code>	<code>std::any</code> ³³³
<code>hpx::any_nons</code>	
<code>hpx::bad_any_cast</code>	<code>std::bad_any_cast</code> ³³⁴
<code>hpx::unique_any_nons</code>	

Functions

Table 2.39: Functions of header `hpx/any.hpp`

Function	C++ standard
<code>hpx::any_cast</code>	<code>std::any_cast</code> ³³⁵
<code>hpx::make_any</code>	<code>std::make_any</code> ³³⁶
<code>hpx::make_any_nons</code>	
<code>hpx::make_unique_any_nons</code>	

hpx/assert.hpp

The header `hpx/assert.hpp`³³⁷ corresponds to the C++ standard library header `cassert`³³⁸.

`HPX_ASSERT` is the *HPX* equivalent to `assert` in `cassert`. `HPX_ASSERT` can also be used in CUDA device code.

Macros

Table 2.40: Macros of header `hpx/assert.hpp`

Macro
<code>HPX_ASSERT</code>
<code>HPX_ASSERT_MSG</code>

³³¹ <http://github.com/STEllAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/any.hpp>

³³² <http://en.cppreference.com/w/cpp/header/any>

³³³ <http://en.cppreference.com/w/cpp/utility/any>

³³⁴ http://en.cppreference.com/w/cpp/utility/any/bad_any_cast

³³⁵ http://en.cppreference.com/w/cpp/utility/any/any_cast

³³⁶ http://en.cppreference.com/w/cpp/utility/any/make_any

³³⁷ <http://github.com/STEllAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/core/assertion/include/hpx/assert.hpp>

³³⁸ <http://en.cppreference.com/w/cpp/header/cassert>

hpx/barrier.hpp

The header `hpx/barrier.hpp`³³⁹ corresponds to the C++ standard library header `barrier`³⁴⁰ 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

Table 2.41: Classes of header `hpx/barrier.hpp`

Class	C++ standard
<code>hpx::barrier</code>	<code>std::barrier</code> ³⁴¹

Table 2.42: Distributed implementation of classes of header `hpx/barrier.hpp`

Class
<code>hpx::distributed::barrier</code>

hpx/channel.hpp

The header `hpx/channel.hpp`³⁴² contains a local and a distributed channel implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

Table 2.43: Classes of header `hpx/channel.hpp`

Class
<code>hpx::channel</code>

Table 2.44: Distributed implementation of classes of header `hpx/channel.hpp`

Class
<code>hpx::distributed::channel</code>

³³⁹ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/barrier.hpp>

³⁴⁰ <http://en.cppreference.com/w/cpp/header/barrier>

³⁴¹ <http://en.cppreference.com/w/cpp/thread/barrier>

³⁴² <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/channel.hpp>

hpx/chrono.hpp

The header `hpx/chrono.hpp`³⁴³ corresponds to the C++ standard library header `chrono`³⁴⁴. The following replacements and extensions are provided compared to `chrono`³⁴⁵.

Classes

Table 2.45: Classes of header `hpx/chrono.hpp`

Class	C++ standard
<code>hpx::chrono::high_resolution_clock</code>	<code>std::high_resolution_clock</code> ³⁴⁶
<code>hpx::chrono::high_resolution_timer</code>	
<code>hpx::chrono::steady_time_point</code>	<code>std::time_point</code> ³⁴⁷

hpx/condition_variable.hpp

The header `hpx/condition_variable.hpp`³⁴⁸ corresponds to the C++ standard library header `condition_variable`³⁴⁹.

Classes

Table 2.46: Classes of header `hpx/condition_variable.hpp`

Class	C++ standard
<code>hpx::condition_variable</code>	<code>std::condition_variable</code> ³⁵⁰
<code>hpx::condition_variable_any</code>	<code>std::condition_variable_any</code> ³⁵¹
<code>hpx::cv_status</code>	<code>std::cv_status</code> ³⁵²

hpx/exception.hpp

The header `hpx/exception.hpp`³⁵³ corresponds to the C++ standard library header `exception`³⁵⁴. `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.

³⁴³ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/chrono.hpp>

³⁴⁴ <http://en.cppreference.com/w/cpp/header/chrono>

³⁴⁵ <http://en.cppreference.com/w/cpp/header/chrono>

³⁴⁶ http://en.cppreference.com/w/cpp/chrono/high_resolution_clock

³⁴⁷ http://en.cppreference.com/w/cpp/chrono/time_point

³⁴⁸ http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/condition_variable.hpp

³⁴⁹ http://en.cppreference.com/w/cpp/header/condition_variable

³⁵⁰ http://en.cppreference.com/w/cpp/thread/condition_variable

³⁵¹ http://en.cppreference.com/w/cpp/thread/condition_variable_any

³⁵² http://en.cppreference.com/w/cpp/thread/cv_status

³⁵³ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/exception.hpp>

³⁵⁴ <http://en.cppreference.com/w/cpp/header/exception>

Macros

- `HPX_THROW_EXCEPTION`

Classes

Table 2.47: Classes of header `hpx/exception.hpp`

Class	C++ standard
<code>hpx::exception</code>	<code>std::exception</code> ³⁵⁵

`hpx/execution.hpp`

The header `hpx/execution.hpp`³⁵⁶ corresponds to the C++ standard library header `execution`³⁵⁷. 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

Table 2.48: Constants of header `hpx/execution.hpp`

Constant	C++ standard
<code>hpx::execution::seq</code>	<code>std::execution_policy_tag</code> ³⁵⁸
<code>hpx::execution::par</code>	<code>std::execution_policy_tag</code> ³⁵⁹
<code>hpx::execution::par_unseq</code>	<code>std::execution_policy_tag</code> ³⁶⁰
<code>hpx::execution::task</code>	

³⁵⁵ <http://en.cppreference.com/w/cpp/error/exception>

³⁵⁶ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/execution.hpp>

³⁵⁷ <http://en.cppreference.com/w/cpp/header/execution>

³⁵⁸ http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag

³⁵⁹ http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag

³⁶⁰ http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag

Classes

Table 2.49: Classes of header `hpx/execution.hpp`

Class	C++ standard
<code>hpx::execution::sequenced_policy</code>	<code>std::execution_policy_tag_t</code> ³⁶¹
<code>hpx::execution::parallel_policy</code>	<code>std::execution_policy_tag_t</code> ³⁶²
<code>hpx::execution::parallel_unsequenced_policy</code>	<code>std::execution_policy_tag_t</code> ³⁶³
<code>hpx::execution::sequenced_task_policy</code>	
<code>hpx::execution::parallel_task_policy</code>	
<code>hpx::execution::auto_chunk_size</code>	
<code>hpx::execution::dynamic_chunk_size</code>	
<code>hpx::execution::guided_chunk_size</code>	
<code>hpx::execution::persistent_auto_chunk_size</code>	
<code>hpx::execution::static_chunk_size</code>	

`hpx/functional.hpp`

The header `hpx/functional.hpp`³⁶⁴ corresponds to the C++ standard library header `functional`³⁶⁵. `hpx::function` is a more efficient and serializable replacement for `std::function`.

Constants

The following constants correspond to the C++ standard `std::placeholders`³⁶⁶

Table 2.50: Constants of header `hpx/functional.hpp`

Constant
<code>hpx::placeholders::_1</code>
<code>hpx::placeholders::_2</code>
<code>...</code>
<code>hpx::placeholders::_9</code>

Classes

Table 2.51: Classes of header `hpx/functional.hpp`

Class	C++ standard
<code>hpx::function</code>	<code>std::function</code> ³⁶⁷
<code>hpx::function_ref</code>	<code>P0792</code> ³⁶⁸
<code>hpx::move_only_function</code>	<code>std::move_only_function</code> ³⁶⁹
<code>hpx::traits::is_bind_expression</code>	<code>std::is_bind_expression</code> ³⁷⁰
<code>hpx::traits::is_placeholder</code>	<code>std::is_placeholder</code> ³⁷¹
<code>hpx::scoped_annotation</code>	

³⁶¹ http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

³⁶² http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

³⁶³ http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

³⁶⁴ <http://github.com/STEllAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/functional.hpp>

³⁶⁵ <http://en.cppreference.com/w/cpp/header/functional>

³⁶⁶ <http://en.cppreference.com/w/cpp/utility/functional/placeholders>

Functions

Table 2.52: Functions of header `hpx/functional.hpp`

Function	C++ standard
<code>hpx::annotated_function</code>	
<code>hpx::bind</code>	<code>std::bind</code> ³⁷²
<code>hpx::experimental::bind_back</code>	
<code>hpx::bind_front</code>	<code>std::bind_front</code> ³⁷³
<code>hpx::invoke</code>	<code>std::invoke</code> ³⁷⁴
<code>hpx::invoke_fused</code>	<code>std::apply</code> ³⁷⁵
<code>hpx::invoke_fused_r</code>	
<code>hpx::mem_fn</code>	<code>std::mem_fn</code> ³⁷⁶

hpx/future.hpp

The header `hpx/future.hpp`³⁷⁷ corresponds to the C++ standard library header `future`³⁷⁸. See *Extended facilities for futures* for more information about extensions to futures compared to the C++ standard library.

This header file also 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.

Classes

Table 2.53: Classes of header `hpx/future.hpp`

Class	C++ standard
<code>hpx::future</code>	<code>std::future</code> ³⁷⁹
<code>hpx::shared_future</code>	<code>std::shared_future</code> ³⁸⁰
<code>hpx::promise</code>	<code>std::promise</code> ³⁸¹
<code>hpx::launch</code>	<code>std::launch</code> ³⁸²
<code>hpx::packaged_task</code>	<code>std::packaged_task</code> ³⁸³

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

³⁶⁷ <http://en.cppreference.com/w/cpp/utility/functional/function>

³⁶⁸ <http://wg21.link/p0792>

³⁶⁹ http://en.cppreference.com/w/cpp/utility/functional/move_only_function

³⁷⁰ http://en.cppreference.com/w/cpp/utility/functional/is_bind_expression

³⁷¹ http://en.cppreference.com/w/cpp/utility/functional/is_placeholder

³⁷² <http://en.cppreference.com/w/cpp/utility/functional/bind>

³⁷³ http://en.cppreference.com/w/cpp/utility/functional/bind_front

³⁷⁴ <http://en.cppreference.com/w/cpp/utility/functional/invoke>

³⁷⁵ <http://en.cppreference.com/w/cpp/utility/apply>

³⁷⁶ http://en.cppreference.com/w/cpp/utility/functional/mem_fn

³⁷⁷ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/future.hpp>

³⁷⁸ <http://en.cppreference.com/w/cpp/header/future>

³⁷⁹ <http://en.cppreference.com/w/cpp/thread/future>

³⁸⁰ http://en.cppreference.com/w/cpp/thread/shared_future

³⁸¹ <http://en.cppreference.com/w/cpp/thread/promise>

³⁸² <http://en.cppreference.com/w/cpp/thread/launch>

³⁸³ http://en.cppreference.com/w/cpp/thread/packaged_task

after a deprecation period.

Table 2.54: Distributed implementation of classes of header `hpx/latch.hpp`

Class
<code>hpx::distributed::promise</code>

Functions

Table 2.55: Functions of header `hpx/future.hpp`

Function	C++ standard
<code>hpx::async</code>	<code>std::async</code> ³⁸⁴
<code>hpx::apply</code>	
<code>hpx::sync</code>	
<code>hpx::dataflow</code>	
<code>hpx::make_future</code>	
<code>hpx::make_shared_future</code>	
<code>hpx::make_ready_future</code>	P0159 ³⁸⁵
<code>hpx::make_ready_future_alloc</code>	
<code>hpx::make_ready_future_at</code>	
<code>hpx::make_ready_future_after</code>	
<code>hpx::make_exceptional_future</code>	P0159 ³⁸⁶
<code>hpx::when_all</code>	P0159 ³⁸⁷
<code>hpx::when_any</code>	P0159 ³⁸⁸
<code>hpx::when_some</code>	
<code>hpx::when_each</code>	
<code>hpx::wait_all</code>	
<code>hpx::wait_any</code>	
<code>hpx::wait_some</code>	
<code>hpx::wait_each</code>	

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)

³⁸⁴ <http://en.cppreference.com/w/cpp/thread/async>

³⁸⁵ <http://wg21.link/p0159>

³⁸⁶ <http://wg21.link/p0159>

³⁸⁷ <http://wg21.link/p0159>

³⁸⁸ <http://wg21.link/p0159>

(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::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);
}
```

(continues on next page)

(continued from previous page)

```

hpx::shared_future<double> f10 = hpx::async([]() { return 123.45; });
hpx::future<hpx::tuple<double, double>> f11 = hpx::dataflow(
    [] (hpx::future<double> a, hpx::future<double> b,
        hpx::shared_future<double> c, double d) {
        return hpx::make_tuple<>(a.get() + b.get(), c.get() / d);
    },
    f8, f9, f10, -3.9);

// split_future gives a tuple of futures from a future of tuple
hpx::tuple<hpx::future<double>, hpx::future<double>> f12 =
    hpx::split_future(std::move(f11));
std::cout << hpx::get<1>(f12).get() << std::endl;

return 0;
}

```

hpx/init.hpp

The header `hpx/init.hpp`³⁸⁹ 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

Table 2.56: Classes of header `hpx/init.hpp`

Class
<code>hpx::init_params</code>
<code>hpx::runtime_mode</code>

Functions

Table 2.57: Functions of header `hpx/init.hpp`

Function
<code>hpx::init</code>
<code>hpx::start</code>
<code>hpx::finalize</code>
<code>hpx::disconnect</code>
<code>hpx::suspend</code>
<code>hpx::resume</code>

³⁸⁹ http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/init_runtime/include/hpx/init.hpp

hpx/latch.hpp

The header `hpx/latch.hpp`³⁹⁰ corresponds to the C++ standard library header `latch`³⁹¹. It contains a local and a distributed latch implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

Table 2.58: Classes of header hpx/latch.hpp

Class	C++ standard
<code>hpx::latch</code>	<code>std::latch</code> ³⁹²

Table 2.59: Distributed implementation of classes of header hpx/latch.hpp

Class
<code>hpx::distributed::latch</code>

hpx/mutex.hpp

The header `hpx/mutex.hpp`³⁹³ corresponds to the C++ standard library header `mutex`³⁹⁴.

Classes

Table 2.60: Classes of header hpx/mutex.hpp

Class	C++ standard
<code>hpx::mutex</code>	<code>std::mutex</code> ³⁹⁵
<code>hpx::no_mutex</code>	
<code>hpx::once_flag</code>	<code>std::once_flag</code> ³⁹⁶
<code>hpx::recursive_mutex</code>	<code>std::recursive_mutex</code> ³⁹⁷
<code>hpx::spinlock</code>	
<code>hpx::timed_mutex</code>	<code>std::timed_mutex</code> ³⁹⁸
<code>hpx::unlock_guard</code>	

³⁹⁰ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/latch.hpp>

³⁹¹ <http://en.cppreference.com/w/cpp/header/latch>

³⁹² <http://en.cppreference.com/w/cpp/thread/latch>

³⁹³ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/mutex.hpp>

³⁹⁴ <http://en.cppreference.com/w/cpp/header/mutex>

³⁹⁵ <http://en.cppreference.com/w/cpp/thread/mutex>

³⁹⁶ http://en.cppreference.com/w/cpp/thread/once_flag

³⁹⁷ http://en.cppreference.com/w/cpp/thread/recursive_mutex

³⁹⁸ http://en.cppreference.com/w/cpp/thread/timed_mutex

Functions

Table 2.61: Functions of header `hpx/mutex.hpp`

Class	C++ standard
<code>hpx::call_once</code>	<code>std::call_once</code> ³⁹⁹

`hpx/memory.hpp`

The header `hpx/memory.hpp`⁴⁰⁰ corresponds to the C++ standard library header `memory`⁴⁰¹. It contains parallel versions of the copy, fill, move, and construct helper functions in `memory`⁴⁰². See [Using parallel algorithms](#) for more information about the parallel algorithms.

Functions

Table 2.62: *hpx* functions of header `hpx/memory.hpp`

<i>hpx</i> function	C++ standard
<code>hpx::uninitialized_copy</code>	<code>std::uninitialized_copy</code> ⁴⁰³
<code>hpx::uninitialized_copy_n</code>	<code>std::uninitialized_copy_n</code> ⁴⁰⁴
<code>hpx::uninitialized_default_construct</code>	<code>std::uninitialized_default_construct</code> ⁴⁰⁵
<code>hpx::uninitialized_default_construct_n</code>	<code>std::uninitialized_default_construct_n</code> ⁴⁰⁶
<code>hpx::uninitialized_fill</code>	<code>std::uninitialized_fill</code> ⁴⁰⁷
<code>hpx::uninitialized_fill_n</code>	<code>std::uninitialized_fill_n</code> ⁴⁰⁸
<code>hpx::uninitialized_move</code>	<code>std::uninitialized_move</code> ⁴⁰⁹
<code>hpx::uninitialized_move_n</code>	<code>std::uninitialized_move_n</code> ⁴¹⁰
<code>hpx::uninitialized_value_construct</code>	<code>std::uninitialized_value_construct</code> ⁴¹¹
<code>hpx::uninitialized_value_construct_n</code>	<code>std::uninitialized_value_construct_n</code> ⁴¹²

³⁹⁹ http://en.cppreference.com/w/cpp/thread/call_once

⁴⁰⁰ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/memory.hpp>

⁴⁰¹ <http://en.cppreference.com/w/cpp/header/memory>

⁴⁰² <http://en.cppreference.com/w/cpp/header/memory>

⁴⁰³ http://en.cppreference.com/w/cpp/memory/uninitialized_copy

⁴⁰⁴ http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n

⁴⁰⁵ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct

⁴⁰⁶ http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n

⁴⁰⁷ http://en.cppreference.com/w/cpp/memory/uninitialized_fill

⁴⁰⁸ http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n

⁴⁰⁹ http://en.cppreference.com/w/cpp/memory/uninitialized_move

⁴¹⁰ http://en.cppreference.com/w/cpp/memory/uninitialized_move_n

⁴¹¹ http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct

⁴¹² http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct_n

Table 2.63: *hpx::ranges* functions of header `hpx/memory.hpp`

<i>hpx::ranges</i> function	C++ standard
<code>hpx::ranges::uninitialized_copy</code>	<code>std::uninitialized_copy</code> ⁴¹³
<code>hpx::ranges::uninitialized_copy_n</code>	<code>std::uninitialized_copy_n</code> ⁴¹⁴
<code>hpx::ranges::uninitialized_default_construct</code>	<code>std::uninitialized_default_construct</code> ⁴¹⁵
<code>hpx::ranges::uninitialized_default_construct_n</code>	<code>std::uninitialized_default_construct_n</code> ⁴¹⁶
<code>hpx::ranges::uninitialized_fill</code>	<code>std::uninitialized_fill</code> ⁴¹⁷
<code>hpx::ranges::uninitialized_fill_n</code>	<code>std::uninitialized_fill_n</code> ⁴¹⁸
<code>hpx::ranges::uninitialized_move</code>	<code>std::uninitialized_move</code> ⁴¹⁹
<code>hpx::ranges::uninitialized_move_n</code>	<code>std::uninitialized_move_n</code> ⁴²⁰
<code>hpx::ranges::uninitialized_value_construct</code>	<code>std::uninitialized_value_construct</code> ⁴²¹
<code>hpx::ranges::uninitialized_value_construct_n</code>	<code>std::uninitialized_value_construct_n</code> ⁴²²

hpx/numeric.hpp

The header `hpx/numeric.hpp`⁴²³ corresponds to the C++ standard library header `numeric`⁴²⁴. See *Using parallel algorithms* for more information about the parallel algorithms.

Functions

Table 2.64: *hpx* functions of header `hpx/numeric.hpp`

<i>hpx</i> function	C++ standard
<code>hpx::adjacent_difference</code>	<code>std::adjacent_difference</code> ⁴²⁵
<code>hpx::exclusive_scan</code>	<code>std::exclusive_scan</code> ⁴²⁶
<code>hpx::inclusive_scan</code>	<code>std::inclusive_scan</code> ⁴²⁷
<code>hpx::reduce</code>	<code>std::reduce</code> ⁴²⁸
<code>hpx::transform_exclusive_scan</code>	<code>std::transform_exclusive_scan</code> ⁴²⁹
<code>hpx::transform_inclusive_scan</code>	<code>std::transform_inclusive_scan</code> ⁴³⁰
<code>hpx::transform_reduce</code>	<code>std::transform_reduce</code> ⁴³¹

⁴¹³ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_copy

⁴¹⁴ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_copy_n

⁴¹⁵ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_default_construct

⁴¹⁶ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_default_construct_n

⁴¹⁷ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_fill

⁴¹⁸ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_fill_n

⁴¹⁹ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_move

⁴²⁰ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_move_n

⁴²¹ http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_value_construct

⁴²² http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_value_construct_n

⁴²³ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/numeric.hpp>

⁴²⁴ <http://en.cppreference.com/w/cpp/header/numeric>

⁴²⁵ http://en.cppreference.com/w/cpp/algorithm/adjacent_difference

⁴²⁶ http://en.cppreference.com/w/cpp/algorithm/exclusive_scan

⁴²⁷ http://en.cppreference.com/w/cpp/algorithm/inclusive_scan

⁴²⁸ <http://en.cppreference.com/w/cpp/algorithm/reduce>

⁴²⁹ http://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan

⁴³⁰ http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan

⁴³¹ http://en.cppreference.com/w/cpp/algorithm/transform_reduce

Table 2.65: *hpx::ranges* functions of header `hpx/numeric.hpp`

<i>hpx::ranges</i> function
<code>hpx::ranges::exclusive_scan</code>
<code>hpx::ranges::inclusive_scan</code>
<code>hpx::ranges::transform_exclusive_scan</code>
<code>hpx::ranges::transform_inclusive_scan</code>

hpx/optional.hpp

The header `hpx/optional.hpp`⁴³² corresponds to the C++ standard library header `optional`⁴³³. `hpx::optional` is compatible with `std::optional`.

Constants

- `hpx::nullopt`

Classes

Table 2.66: Classes of header `hpx/optional.hpp`

Class	C++ standard
<code>hpx::optional</code>	<code>std::optional</code> ⁴³⁴
<code>hpx::nullopt_t</code>	<code>std::nullopt_t</code> ⁴³⁵
<code>hpx::bad_optional_access</code>	<code>std::bad_optional_access</code> ⁴³⁶

hpx/runtime.hpp

The header `hpx/runtime.hpp`⁴³⁷ contains functions for accessing local and distributed runtime information.

Typedefs

Table 2.67: Typedefs of header `hpx/runtime.hpp`

Typedef
<code>hpx::startup_function_type</code>
<code>hpx::shutdown_function_type</code>

⁴³² <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/optional.hpp>

⁴³³ <http://en.cppreference.com/w/cpp/header/optional>

⁴³⁴ <http://en.cppreference.com/w/cpp/utility/optional>

⁴³⁵ http://en.cppreference.com/w/cpp/utility/nullopt_t

⁴³⁶ http://en.cppreference.com/w/cpp/utility/optional/bad_optional_access

⁴³⁷ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/runtime.hpp>

Functions

Table 2.68: Functions of header `hpx/runtime.hpp`

Function
<code>hpx::find_root_locality</code>
<code>hpx::find_all_localities</code>
<code>hpx::find_remote_localities</code>
<code>hpx::find_locality</code>
<code>hpx::get_colocation_id</code>
<code>hpx::get_locality_id</code>
<code>hpx::get_num_worker_threads</code>
<code>hpx::get_worker_thread_num</code>
<code>hpx::get_thread_name</code>
<code>hpx::register_pre_startup_function</code>
<code>hpx::register_startup_function</code>
<code>hpx::register_pre_shutdown_function</code>
<code>hpx::register_shutdown_function</code>
<code>hpx::get_num_localities</code>
<code>hpx::get_locality_name</code>

`hpx/source_location.hpp`

The header `hpx/source_location.hpp`⁴³⁸ corresponds to the C++ standard library header `source_location`⁴³⁹.

Classes

Table 2.69: Classes of header `hpx/system_error.hpp`

Class	C++ standard
<code>hpx::source_location</code>	<code>std::source_location</code> ⁴⁴⁰

`hpx/system_error.hpp`

The header `hpx/system_error.hpp`⁴⁴¹ corresponds to the C++ standard library header `system_error`⁴⁴².

⁴³⁸ http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/source_location.hpp

⁴³⁹ http://en.cppreference.com/w/cpp/header/source_location

⁴⁴⁰ http://en.cppreference.com/w/cpp/utility/source_location

⁴⁴¹ http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/system_error.hpp

⁴⁴² http://en.cppreference.com/w/cpp/header/system_error

Classes

Table 2.70: Classes of header `hpx/system_error.hpp`

Class	C++ standard
<code>hpx::error_code</code>	<code>std::error_code</code> ⁴⁴³

`hpx/task_block.hpp`

The header `hpx/task_block.hpp`⁴⁴⁴ corresponds to the `task_block` feature in N4411⁴⁴⁵. See [Using task blocks](#) for more details on using task blocks.

Classes

Table 2.71: Classes of header `hpx/task_block.hpp`

Class
<code>hpx::parallel::v2::task_canceled_exception</code>
<code>hpx::parallel::v2::task_block</code>

Functions

Table 2.72: Functions of header `hpx/task_block.hpp`

Function
<code>hpx::parallel::v2::define_task_block</code>
<code>hpx::parallel::v2::define_task_block_restore_thread</code>

`hpx/thread.hpp`

The header `hpx/thread.hpp`⁴⁴⁶ corresponds to the C++ standard library header `thread`⁴⁴⁷. 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.

⁴⁴³ http://en.cppreference.com/w/cpp/error/error_code

⁴⁴⁴ http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/task_block.hpp

⁴⁴⁵ <http://wg21.link/n4411>

⁴⁴⁶ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/thread.hpp>

⁴⁴⁷ <http://en.cppreference.com/w/cpp/header/thread>

Classes

Table 2.73: Classes of header hpx/thread.hpp

Class	C++ standard
<code>hpx::thread</code>	<code>std::thread</code> ⁴⁴⁸
<code>hpx::jthread</code>	<code>std::jthread</code> ⁴⁴⁹

Functions

Table 2.74: Functions of header hpx/thread.hpp

Function	C++ standard
<code>hpx::this_thread::yield</code>	<code>std::yield</code> ⁴⁵⁰
<code>hpx::this_thread::get_id</code>	<code>std::get_id</code> ⁴⁵¹
<code>hpx::this_thread::sleep_for</code>	<code>std::sleep_for</code> ⁴⁵²
<code>hpx::this_thread::sleep_until</code>	<code>std::sleep_until</code> ⁴⁵³

hpx/semaphore.hpp

The header `hpx/semaphore.hpp`⁴⁵⁴ corresponds to the C++ standard library header `semaphore`⁴⁵⁵.

Classes

Table 2.75: Classes of header hpx/semaphore.hpp

Class	C++ standard
<code>hpx::binary_semaphore</code>	<code>std::counting_semaphore</code> ⁴⁵⁶
<code>hpx::counting_semaphore</code>	<code>std::counting_semaphore</code> ⁴⁵⁷

⁴⁴⁸ <http://en.cppreference.com/w/cpp/thread/thread>

⁴⁴⁹ <http://en.cppreference.com/w/cpp/thread/jthread>

⁴⁵⁰ <http://en.cppreference.com/w/cpp/thread/yield>

⁴⁵¹ http://en.cppreference.com/w/cpp/thread/get_id

⁴⁵² http://en.cppreference.com/w/cpp/thread/sleep_for

⁴⁵³ http://en.cppreference.com/w/cpp/thread/sleep_until

⁴⁵⁴ <http://github.com/STEllAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/semaphore.hpp>

⁴⁵⁵ <http://en.cppreference.com/w/cpp/header/semaphore>

⁴⁵⁶ http://en.cppreference.com/w/cpp/thread/counting_semaphore

⁴⁵⁷ http://en.cppreference.com/w/cpp/thread/counting_semaphore

hpx/shared_mutex.hpp

The header `hpx/shared_mutex.hpp`⁴⁵⁸ corresponds to the C++ standard library header `shared_mutex`⁴⁵⁹.

Classes

Table 2.76: Classes of header `hpx/shared_mutex.hpp`

Class	C++ standard
<code>hpx::shared_mutex</code>	<code>std::shared_mutex</code> ⁴⁶⁰

hpx/stop_token.hpp

The header `hpx/stop_token.hpp`⁴⁶¹ corresponds to the C++ standard library header `stop_token`⁴⁶².

Constants

Table 2.77: Constants of header `hpx/stop_token.hpp`

Constant	C++ standard
<code>hpx::nostopstate</code>	<code>std::nostopstate</code> ⁴⁶³

Classes

Table 2.78: Classes of header `hpx/stop_token.hpp`

Class	C++ standard
<code>hpx::stop_callback</code>	<code>std::stop_callback</code> ⁴⁶⁴
<code>hpx::stop_source</code>	<code>std::stop_source</code> ⁴⁶⁵
<code>hpx::stop_token</code>	<code>std::stop_token</code> ⁴⁶⁶
<code>hpx::nostopstate_t</code>	<code>std::nostopstate_t</code> ⁴⁶⁷

⁴⁵⁸ http://github.com/STEllAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/shared_mutex.hpp

⁴⁵⁹ http://en.cppreference.com/w/cpp/header/shared_mutex

⁴⁶⁰ http://en.cppreference.com/w/cpp/thread/shared_mutex

⁴⁶¹ http://github.com/STEllAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/stop_token.hpp

⁴⁶² http://en.cppreference.com/w/cpp/header/stop_token

⁴⁶³ http://en.cppreference.com/w/cpp/thread/stop_source/nostopstate

⁴⁶⁴ http://en.cppreference.com/w/cpp/thread/stop_callback

⁴⁶⁵ http://en.cppreference.com/w/cpp/thread/stop_source

⁴⁶⁶ http://en.cppreference.com/w/cpp/thread/stop_token

⁴⁶⁷ http://en.cppreference.com/w/cpp/thread/stop_source/nostopstate_t

hpx/tuple.hpp

The header `hpx/tuple.hpp`⁴⁶⁸ corresponds to the C++ standard library header `tuple`⁴⁶⁹. `hpx::tuple` can be used in CUDA device code, unlike `std::tuple`.

Constants

Table 2.79: Constants of header `hpx/tuple.hpp`

Constant	C++ standard
<code>hpx::ignore</code>	<code>std::ignore</code> ⁴⁷⁰

Classes

Table 2.80: Classes of header `hpx/tuple.hpp`

Class	C++ standard
<code>hpx::tuple</code>	<code>std::tuple</code> ⁴⁷¹
<code>hpx::tuple_size</code>	<code>std::tuple_size</code> ⁴⁷²
<code>hpx::tuple_element</code>	<code>std::tuple_element</code> ⁴⁷³

Functions

Table 2.81: Functions of header `hpx/tuple.hpp`

Function	C++ standard
<code>hpx::make_tuple</code>	<code>std::tuple_element</code> ⁴⁷⁴
<code>hpx::tie</code>	<code>std::tie</code> ⁴⁷⁵
<code>hpx::forward_as_tuple</code>	<code>std::forward_as_tuple</code> ⁴⁷⁶
<code>hpx::tuple_cat</code>	<code>std::tuple_cat</code> ⁴⁷⁷
<code>hpx::get</code>	<code>std::get</code> ⁴⁷⁸

⁴⁶⁸ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/tuple.hpp>

⁴⁶⁹ <http://en.cppreference.com/w/cpp/header/tuple>

⁴⁷⁰ <http://en.cppreference.com/w/cpp/utility/tuple/ignore>

⁴⁷¹ <http://en.cppreference.com/w/cpp/utility/tuple>

⁴⁷² http://en.cppreference.com/w/cpp/utility/tuple_size

⁴⁷³ http://en.cppreference.com/w/cpp/utility/tuple_element

⁴⁷⁴ http://en.cppreference.com/w/cpp/utility/tuple/tuple_element

⁴⁷⁵ <http://en.cppreference.com/w/cpp/utility/tuple/tie>

⁴⁷⁶ http://en.cppreference.com/w/cpp/utility/tuple/forward_as_tuple

⁴⁷⁷ http://en.cppreference.com/w/cpp/utility/tuple/tuple_cat

⁴⁷⁸ <http://en.cppreference.com/w/cpp/utility/tuple/get>

hpx/type_traits.hpp

The header `hpx/type_traits.hpp`⁴⁷⁹ corresponds to the C++ standard library header `type_traits`⁴⁸⁰.

Classes

Table 2.82: Classes of header `hpx/type_traits.hpp`

Class	C++ standard
<code>hpx::is_invocable</code>	<code>std::is_invocable</code> ⁴⁸¹
<code>hpx::is_invocable_r</code>	<code>std::is_invocable</code> ⁴⁸²

hpx/unwrap.hpp

The header `hpx/unwrap.hpp`⁴⁸³ contains utilities for unwrapping futures.

Classes

Table 2.83: Classes of header `hpx/unwrap.hpp`

Class
<code>hpx::functional::unwrap</code>
<code>hpx::functional::unwrap_n</code>
<code>hpx::functional::unwrap_all</code>

Functions

Table 2.84: Functions of header `hpx/unwrap.hpp`

Function
<code>hpx::unwrap</code>
<code>hpx::unwrap_n</code>
<code>hpx::unwrap_all</code>
<code>hpx::unwrapping</code>
<code>hpx::unwrapping_n</code>
<code>hpx::unwrapping_all</code>

⁴⁷⁹ http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/type_traits.hpp

⁴⁸⁰ http://en.cppreference.com/w/cpp/header/type_traits

⁴⁸¹ http://en.cppreference.com/w/cpp/types/is_invocable

⁴⁸² http://en.cppreference.com/w/cpp/types/is_invocable

⁴⁸³ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/full/include/include/hpx/unwrap.hpp>

hpx/version.hpp

The header `hpx/version.hpp`⁴⁸⁴ provides version information about *HPX*.

Macros

Table 2.85: Macros of header `hpx/version.hpp`

Macro
<code>HPX_VERSION_MAJOR</code>
<code>HPX_VERSION_MINOR</code>
<code>HPX_VERSION_SUBMINOR</code>
<code>HPX_VERSION_FULL</code>
<code>HPX_VERSION_DATE</code>
<code>HPX_VERSION_TAG</code>
<code>HPX_AGAS_VERSION</code>

Functions

Table 2.86: Functions of header `hpx/version.hpp`

Function
<code>hpx::major_version</code>
<code>hpx::minor_version</code>
<code>hpx::subminor_version</code>
<code>hpx::full_version</code>
<code>hpx::full_version_as_string</code>
<code>hpx::tag</code>
<code>hpx::agas_version</code>
<code>hpx::build_type</code>
<code>hpx::build_date_time</code>

hpx/wrap_main.hpp

The header `hpx/wrap_main.hpp`⁴⁸⁵ 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.

⁴⁸⁴ <http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/libs/core/version/include/hpx/version.hpp>

⁴⁸⁵ http://github.com/STELLAR-GROUP/hpx/blob/0ac7ec67ce1672e16745696450c6c8b09d643e3c/include/hpx/wrap_main.hpp

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.

algorithms

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parallel/task_block.hpp

See *Public API* for a list of names and headers that are part of the public *HPX* API.

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.

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

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.

Throws `exception_list` – 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.

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

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.

Throws *exception_list* – 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.

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

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.

Throws *exception_list* – specified in Exception Handling.

```
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.

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

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.

Throws `exception_list` – specified in Exception Handling.

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(); });
    });
    ↪active define_task_block([&](auto& tr) {
        tr.run(f());
        /// ...
    });
    /// ...
});
```

Template Parameters `ExPolicy` – The execution policy an instance of a `task_block` was created with. This defaults to `parallel_policy`.

Public Types

using **execution_policy** = *ExPolicy*

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

Public Functions

inline *execution_policy* const &**get_execution_policy()** const

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

template<typename **F**, typename ...**Ts**>

inline 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.

Throws `task_canceled_exception` – described in Exception Handling.

template<typename **Executor**, typename **F**, typename ...**Ts**>

inline 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.

Throws `task_canceled_exception` – described in Exception Handling. The function will also throw an `exception_list` holding all exceptions that were caught while executing the tasks.

inline 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); });
    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.

Throws 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.

inline `ExPolicy &policy()`

Returns a reference to the execution policy used to construct this object.

Precondition: this shall be the active `task_block`.

inline `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

```
inline task_canceled_exception() noexcept
```

hpx/parallel/task_group.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **execution**

 namespace **experimental**

 class **task_group**

Public Functions

```
task_group()
```

```
~task_group()
```

```
template<typename Executor, typename F, typename ...Ts>
inline void run(Executor &&exec, F &&f, Ts&&... ts)
```

```
template<typename F, typename ...Ts>
inline 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 class serialization::access  
  
struct on_exit
```

Public Functions

```
explicit on_exit(task_group &tg)  
  
~on_exit()  
  
on_exit(on_exit const &rhs) = delete  
  
on_exit &operator=(on_exit const &rhs) = delete  
  
on_exit(on_exit &&rhs) noexcept  
  
on_exit &operator=(on_exit &&rhs) noexcept
```

Public Members

```
hpx::lcos::local::latch *latch_
```

hpx/parallel/algorithms/adjacent_difference.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter1, typename FwdIter2>
FwdIter2 adjacent_difference(FwdIter1 first, FwdIter1 last, FwdIter2 dest)
```

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.

Note: Complexity: Exactly (last - first) - 1 application of the binary operator and (last - first) assignments.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the input range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of a forward iterator.

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.
- **dest** – Refers to the beginning of the sequence of elements the results will be assigned to.

Returns The *adjacent_difference* algorithm returns a *FwdIter2*. The *adjacent_difference* algorithm returns an iterator to the element past the last element written.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> adjacent_difference(ExPolicy
    &&policy,
    FwdIter1 first,
    FwdIter1 last,
    FwdIter2 dest)
```

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. Executed according to the policy.

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.

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.

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 a forward iterator.
- **FwdIter2** – The type of the source iterators used for the output 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 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.

Returns 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_difference* algorithm returns an iterator to the element past the last element written.

```
template<typename FwdIter1, typename FwdIter2, typename Op>
FwdIter2 adjacent_difference(FwdIter1 first, FwdIter1 last, FwdIter2 dest, Op &&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

Note: Complexity: Exactly (last - first) - 1 application of the binary operator and (last - first) assignments.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the input range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of a 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

- **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:

<code>bool op(const Type1 &a, const Type1 &b);</code>

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*.

Returns The *adjacent_difference* algorithm returns *FwdIter2*. The *adjacent_difference* algorithm returns an iterator to the element past the last element written.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> adjacent_difference(ExPolicy
    &&policy,
    FwdIter1 first,
    FwdIter1 last,
    FwdIter2 dest,
    Op &&op)
```

Assigns each value in the range given by *result* its corresponding element in the range [first, last] and the one preceding it except **result*, which is assigned **first*

The 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.

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.

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

Returns 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_difference` algorithm returns an iterator to the element past the last element written.

hpx/parallel/algorithms/adjacent_find.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter, typename Pred = hpx::parallel::v1::detail::equal_to>
InIter adjacent_find(InIter first, InIter 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

Template Parameters

- **InIter** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an input 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 `InIter` can be dereferenced and then implicitly converted to `Type1`.

Returns The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is returned.

```
template<typename ExPolicy, typename FwdIter, typename Pred = hpx::parallel::v1::detail::equal_to>
```

```
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type adjacent_find(ExPolicy &&policy,  
                           FwdIter first, FwdIter last,  
                           Pred &&pred = Pred())
```

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.

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*.

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 a 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*.

Returns The *adjacent_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 *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

hpx/parallel/algorithms/all_any_none.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, bool>::type none_of(ExPolicy &&policy, FwdIter first, FwdIter last,
F &&f)
```

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.

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.

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*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence 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*.

Returns 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 InIter, typename F>
bool none_of(InIter first, InIter last, F &&f)
```

Checks if unary predicate *f* returns true for no elements in the range [first, last).

Note: Complexity: At most *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.
- **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*.

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:

```
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*.

Returns The *none_of* algorithm returns a `bool`. 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>
util::detail::algorithm_result<ExPolicy, bool>::type any_of(ExPolicy &&policy, FwdIter first, FwdIter last, F &&f)
```

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.

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.

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*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence 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*.

Returns 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 InIter, typename F>
bool any_of(InIter first, InIter last, F &&f)
```

Checks if unary predicate *f* returns true for at least one element in the range [first, last).

Note: Complexity: At most *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.
- **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*.

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:

```
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*.

Returns The *any_of* algorithm returns a *bool*. 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>
util::detail::algorithm_result<ExPolicy, bool>::type all_of(ExPolicy &&policy, FwdIter first, FwdIter last, F
&&f)
```

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.

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.

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*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence 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*.

Returns The *all_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

```
template<typename ExPolicy, typename InIter, typename F>
bool all_of(InIter first, InIter last, F &&f)
```

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

Note: Complexity: At most *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.
- **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*.

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:

```
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*.

Returns The *all_of* algorithm returns a *bool*. 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.

hpx/parallel/algorithms/copy.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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*. Executed according to the policy.

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.

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.

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.

Returns 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 FwdIter1, typename FwdIter2>
FwdIter2 copy(FwdIter1 first, FwdIter1 last, FwdIter2 dest)
```

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*.

Note: Complexity: Performs exactly *last - first* assignments.

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 (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.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns a *FwdIter2*. 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. Executed according to the policy.

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.

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.

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.

Returns 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_n* algorithm returns Iterator in the destination range, pointing past the last element copied if *count*>0 or result otherwise.

```
template<typename FwdIter1, typename Size, typename FwdIter2>
FwdIter2 copy_n(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.

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 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

- **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.

Returns The *copy_n* algorithm returns a *FwdIter2*. The *copy_n* algorithm returns Iterator in the destination range, pointing past the last element copied if *count*>0 or result otherwise.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred>
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. Executed according to the policy.

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.

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.

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*.

Returns 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_if* algorithm returns output iterator to the element in the destination range, one past the last element copied.

```
template<typename FwdIter1, typename FwdIter2, typename Pred>
FwdIter2 copy_if(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.

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 (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

- **first** – Refers to the beginning of the sequence 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*.

Returns The *copy_if* algorithm returns a *FwdIter2*. The *copy_if* algorithm returns output iterator to the element in the destination range, one past the last element copied.

hpx/parallel/algorithms/count.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter, typename T>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIter>::difference_type>::type count(ExPolicy
    &&policy,
    FwdIter
    first,
    FwdIter
    last,
    T
    const
    &value)
```

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*. Executed according to the policy.

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **FwdIter** – The type of the source iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to search for (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 search for.

Returns 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 **InIter**, typename **T**>
`std::iterator_traits<InIter>::difference_type count(InIter first, InIter last, T const &value)`

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`.

Note: Complexity: Performs exactly `last - first` comparisons.

Template Parameters

- **InIter** – The type of the source iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to search for (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 search for.

Returns The `count` algorithm returns a `difference_type` (where `difference_type` is defined by `std::iterator_traits<InIter>::difference_type`. The `count` algorithm returns the number of elements satisfying the given criteria.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>

```
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIter>::difference_type>::type count_if(ExPolicy
&&policy,
FwdIter first,
FwdIter last,
F
&&f)
```

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true. Executed according to the policy.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **FwdIter** – The type of the source begin 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*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Returns 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<typename InIter, typename F>
std::iterator_traits<InIter>::difference_type count_if(InIter first, InIter last, F &&f)
```

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.

Template Parameters

- **InIter** – The type of the source begin 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). Unlike its sequential form, the parallel overload of `count_if` requires `F` 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.
- **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`.

Returns The `count_if` algorithm returns `difference_type` (where a `difference_type` is defined by `std::iterator_traits<InIter>::difference_type`. The `count` algorithm returns the number of elements satisfying the given criteria.

hpx/parallel/algorithms/destroy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

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

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last). Executed according to the policy.

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.

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.

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.

Returns 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 FwdIter>
void destroy(FwdIter first, FwdIter last)
```

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

Note: Complexity: Performs exactly *last - first* operations.

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.

Returns The *destroy* algorithm returns a *void*

```
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). Executed according to the policy.

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.

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.

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.

Returns The *destroy_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename FwdIter, typename Size>
FwdIter destroy_n(FwdIter first, Size count)
```

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, first + count).

Note: Complexity: Performs exactly *count* operations, 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 this algorithm 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.

Returns The *destroy_n* algorithm returns a *FwdIter*. The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

hpx/parallel/algorithms/ends_with.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter1, typename InIter2, typename Pred>
bool ends_with(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Pred &&pred)
```

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.

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.

Returns 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>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy &&policy, FwdIter1
    first1, FwdIter1 last1, FwdIter2
    first2, FwdIter2 last2, Pred
    &&pred)
```

Checks whether the second range defined by [*first1*, *last1*) matches the suffix of the first range defined by [*first2*, *last2*). Executed according to the policy.

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.

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.

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.

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

Returns 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.

hpx/parallel/algorithms/equal.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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. Executed according to the policy.

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.

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: Complexity: $O(\min(last1 - first1, last2 - first2))$ applications of the predicate *op*.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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

Returns 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>
util::detail::algorithm_result<ExPolicy, bool>::type equal(ExPolicy &&policy, FwdIter1 first1, FwdIter1
last1, FwdIter2 first2, FwdIter2 last2)
```

Returns true if the range `[first1, last1)` is equal to the range `[first2, last2)`, and false otherwise. Executed according to policy.

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.

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: Complexity: $O(\min(last1 - first1, last2 - first2))$ applications of the predicate `std::equal_to`.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

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.

Returns 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. Executed according to policy.

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.

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: Complexity: $O(\min(last1 - first1, last2 - first2))$ applications of the predicate *op*.

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 \equiv to determine if two elements are equal.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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

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

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2ExPolicy, bool>::type equal(ExPolicy &&policy, FwdIter1 first1, FwdIter1
last1, FwdIter2 first2)
```

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise. Executed according to policy.

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.

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: Complexity: At most $last1 - first1$ applications of the predicate `op`.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

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.

Returns 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 FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
bool equal(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.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *op*.

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.

Template Parameters

- **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

- **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

Returns The `equal` algorithm returns a `bool`. 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 FwdIter1, typename FwdIter2>
bool equal(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2)
```

Returns true if the range `[first1, last1)` is equal to the range `[first2, last2)`, and false otherwise.

Note: Complexity: At most `min(last1 - first1, last2 - first2)` applications of the predicate `std::equal_to`.

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.

Template Parameters

- **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.

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.

Returns The `equal` algorithm returns a `bool`. 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 FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
bool equal(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, Pred &&op = Pred())
```

Returns true if the range `[first1, last1)` is equal to the range `[first2, first2 + (last1 - first1))`, and false otherwise.

Note: Complexity: At most `last1 - first1` applications of the predicate `op`.

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.

Template Parameters

- **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

- **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

Returns The *equal* algorithm returns a *bool*. 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.

hpx/parallel/algorithms/exclusive_scan.hpp

See *Public API* for a list of names and headers that are part of the public *HPX* API.

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.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the i th input element in the i th sum.

Note: Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

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 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.

Returns 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.

```
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.

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.

Note: Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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.

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.

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.

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

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 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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op, typename T>
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.

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.

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

Note: GENERALIZED_NONCOMMUTATIVE_SUM(*op*, a_1, \dots, a_N) is defined as:

- a_1 when N is 1
 - $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_M, \dots, a_N))$ where $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **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.
- **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.
- **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.

Returns 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.

hpx/parallel/algorithms/fill.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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). Executed according to the policy.

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.

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.

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.

Returns 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 FwdIter, typename T>
void fill(FwdIter first, FwdIter last, T value)
```

Assigns the given value to the elements in the range [first, last).

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.
- **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.

Returns The *fill* algorithm returns a `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. Executed according to the policy.

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.

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.

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 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.

Returns The *fill_n* algorithm returns a `hpx::future<void>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*).

```
template<typename FwdIter, typename Size, typename T>
FwdIter fill_n(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.

Note: Complexity: Performs exactly *count* assignments, for *count* > 0.

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.

Returns The *fill_n* algorithm returns a *FwdIter*.

hpx/parallel/algorithms/find.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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. Executed according to the policy.

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.

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.

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

Returns 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 InIter, typename T>
InIter find(InIter first, InIter last, T const &val)
```

Returns the first element in the range [first, last) that is equal to value. Executed according to the policy.

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **InIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input iterator.

- **T** – The type of the value to find (deduced).

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.
- **val** – the value to compare the elements to

Returns The *find* algorithm returns a *InIter*. 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. Executed according to the policy.

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.

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.

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*.

Returns 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 InIter, typename F>
InIter find_if(InIter first, InIter last, F &&f)
```

Returns the first element in the range [first, last) for which predicate *f* returns true.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **InIter** – The type of the source iterators used for the first range (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 *equal* requires *F* to meet the requirements of *CopyConstructible*.

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.
- **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 *InIter* can be dereferenced and then implicitly converted to *Type*.

Returns The *find_if* algorithm returns a *InIter*. 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. Executed according to the policy.

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.

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.

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*.

Returns 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 FwdIter, typename F>
FwdIter find_if_not(FwdIter first, FwdIter last, F &&f)
```

Returns the first element in the range [first, last) for which predicate *f* returns false.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **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

- **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`.

Returns The `find_if_not` algorithm returns a `FwdIter`. 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 = std::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 `op` to compare elements. Executed according to the policy.

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.

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`.

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<>`

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.

Returns 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 FwdIter2ExPolicy, FwdIter1>::type find_end(ExPolicy &&policy, FwdIter1 first1,
FwdIter1 last1, FwdIter2 first2, FwdIter2
last2)
```

Returns the last subsequence of elements `[first2, last2]` found in the range `[first, last]`. Elements are compared using `operator==`. Executed according to the policy.

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.

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.

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.

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.

Returns 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 FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
FwdIter1 find_end(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 *op* to compare elements.

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

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

- **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<>*

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 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.

Returns The *find_end* algorithm returns a *FwdIter1*. 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 FwdIter1, typename FwdIter2>
FwdIter1 find_end(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2)
```

Returns the last subsequence of elements [first2, last2] found in the range [first, last). Elements are compared using *operator==*.

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

- **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.

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 the algorithm will be searching for.
- **last2** – Refers to the end of the sequence of elements of the algorithm will be searching for.

Returns The *find_end* algorithm returns a *FwdIter1*. 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 *op* to compare elements. Executed according to the policy.

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.

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*.

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<>`

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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
util::detail::algorithm_result<ExPolicy, FwdIter1>::type find_first_of(ExPolicy &&policy, FwdIter1 first,
FwdIter1 last, FwdIter2 s_first,
FwdIter2 s_last)
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Elements are compared using `operator==`. Executed according to the policy.

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.

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.

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.

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.

Returns 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.

```
template<typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
FwdIter1 find_first_of(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 *op* to compare elements.

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **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

- **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.

Returns The *find_first_of* algorithm returns a *FwdIter1*. 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.

```
template<typename FwdIter1, typename FwdIter2>
FwdIter1 find_first_of(FwdIter1 first, FwdIter1 last, FwdIter2 s_first, FwdIter2 s_last)
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Elements are compared using *operator==*.

Note: Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- **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.

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.

Returns The *find_first_of* algorithm returns a *FwdIter1*. 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.

hpx/parallel/algorithms/for_each.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

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

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

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 *InIter* can be dereferenced and then implicitly converted to *Type*.

Returns *f*.

```
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). Executed according to the policy.

If *f* returns a result, the result is ignored.

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.

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.

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

Template Parameters

- **ExPolicy** – The type of the execution 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 a 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*.

Returns 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.

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

Note: Complexity: Applies f exactly $count$ times.

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 *InIter* can be dereferenced and then implicitly converted to *Type*.

Returns $first + count$ for non-negative values of *count* and *first* for negative values.

```
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. Executed according to the policy.

If f returns a result, the result is ignored.

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_n* 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.

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.

Note: Complexity: Applies f exactly *count* times.

Template Parameters

- **ExPolicy** – The type of the execution 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_n` 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*.

Returns 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.

hpx/parallel/algorithms/for_loop.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **experimental**

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*.

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:

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.

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.

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.

```
template<typename ExPolicy, typename I, typename ...Args>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy> 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. Executed according to the 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*.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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*.

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:

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.

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.

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.

```
template<typename ExPolicy, typename I, typename S, typename ...Args>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy> 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. Executed according to the 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`.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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`.

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:

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.

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.

Template Parameters

- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **Args** – A parameter pack, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **size** – Refers to the number of items the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last] should expose a signature equivalent to:

```
<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.

```
template<typename ExPolicy, typename I, typename Size, typename ...Args>
hp::parallel::util::detail::algorithm_result_t<ExPolicy> 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. Executed according to the 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`.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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*.

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:

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.

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.

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.

```
template<typename ExPolicy, typename I, typename Size, typename S, typename ...Args>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy> 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. Executed according to the 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*.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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, 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.
- **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.

Returns 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.

hpx/parallel/algorithms/for_loop_induction.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **experimental**

Functions

```
template<typename T>
constexpr hpx::parallel::v2::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.

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)

Returns 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.

namespace **parallel**

namespace **v2**

hpx/parallel/algorithms/for_loop_reduction.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **experimental**

Functions

```
template<typename T, typename Op>
constexpr hpx::parallel::v2::detail::reduction_helper<T, std::decay_t<Op>> reduction(T &var, T const
&identity, Op
&&combiner)
```

The function template returns a reduction object of unspecified type having a value type and encapsulating an identity value for the reduction, a combiner function object, and a live-out object from which the initial value is obtained and into which the final value is stored.

A parallel algorithm uses reduction objects by allocating an unspecified number of instances, called views, of the reduction's value type. Each view is initialized with the reduction object's identity value, except that the live-out object (which was initialized by the caller) comprises one of the views. The algorithm passes a reference to a view to each application of an element-access function, ensuring that no two concurrently-executing invocations share the same view. A view can be shared between two applications that do not execute concurrently, but initialization is performed only once per view.

Modifications to the view by the application of element access functions accumulate as partial results. At some point before the algorithm returns, the partial results are combined, two at a time, using the reduction object's combiner operation until a single value remains, which is then assigned back to the live-out object.

T shall meet the requirements of *CopyConstructible* and *MoveAssignable*. The expression

```
var = combiner(var, var)
```

shall be well formed.

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.

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.

Returns 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 **parallel**

namespace **v2**

hpx/parallel/algorithms/generate.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter, typename F>
hpx::parallel::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*. Executed according to the policy.

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.

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.

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*.

Returns The *generate* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise.

```
template<typename FwdIter, typename F>
FwdIter generate(FwdIter first, FwdIter last, F &&f)
```

Assign each element in range $[first, last]$ a value generated by the given function object *f*.

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

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). Unlike its sequential form, the parallel overload of *equal* requires *F* 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.
- **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*.

Returns The *generate* algorithm returns a *FwdIter*.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename FExPolicy, 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 *f*. Executed according to the policy.

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.

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.

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.
- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **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*.

Returns The *generate_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. *generate_n* algorithm returns iterator one past the last element assigned if *count*>0, *first* otherwise.

```
template<typename FwdIter, typename Size, typename F

```

FwdIter **generate_n**(*FwdIter* first, *Size* count, *F* &&*f*)

Assigns each element in range [*first*, *first*+*count*) a value generated by the given function object *f*.

Note: Complexity: Exactly *count* invocations of *f* and assignments, for *count* > 0.

Template Parameters

- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **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

- **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*.

Returns The *generate_n* algorithm returns a *FwdIter*. *generate_n* algorithm returns iterator one past the last element assigned if *count*>0, first otherwise.

hpx/parallel/algorithms/includes.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = hpx::parallel::v1::detail::less>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type::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*. Executed according to the policy.

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.

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.

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

Returns 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 **FwdIter1**, typename **FwdIter2**, typename **Pred** = `hpx::parallel::v1::detail::less`>

bool **includes**(*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*.

Note: At most $2*(N1+N2-1)$ comparisons, where $N1 = \text{std}::\text{distance}(\text{first1}, \text{last1})$ and $N2 = \text{std}::\text{distance}(\text{first2}, \text{last2})$.

Template Parameters

- **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 *Copy-Constructible*. This defaults to *std::less<>*

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **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

Returns The *includes* algorithm returns a *bool*. 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.

hpx/parallel/algorithms/inclusive_scan.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the *i*th input element in the *i*th sum.

Note: Complexity: O(*last - first*) applications of the predicate *op*, here std::plus<>().

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where 1 < K+1 = M <= N.

Template Parameters

- **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.

Returns 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.

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

```
hpx::parallel::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, ..., *(first + (i - result))). Executed according to the policy.

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.

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.

Note: Complexity: O(*last - first*) applications of the predicate *op*, here std::plus<>().

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **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.

Returns 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.

```
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.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the i th input element in the i th sum.

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

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 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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type inclusive_scan(ExPolicy &&policy,  
                                FwdIter1 first,  
                                FwdIter1 last,  
                                FwdIter2 dest, Op  
                                &&op)
```

Assigns through each iterator *i* in [*result*, *result* + (*last* - *first*)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(*op*, **first*, ..., *(*first* + (*i* - *result*))). Executed according to the policy.

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.

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.

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

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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.

template<typename **InIter**, typename **OutIter**, typename **Op**, typename **T**>
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.

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.

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

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 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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op, typename T>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type inclusive_scan(ExPolicy &&policy,
                                         FwdIter1 first,
                                         FwdIter1 last,
                                         FwdIter2 dest, Op
                                         &&op, T init)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, \dots, *(first + (i - result)))`. Executed according to the policy.

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.

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.

Note: Complexity: $O(last - first)$ applications of the predicate `op`.

Note: `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, 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 $1 < K+1 = M \leq N$.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **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.

- **init** – The initial value for the generalized sum.

Returns 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.

hpx/parallel/algorithms/is_heap.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename RandIter, typename Comp = hpx::parallel::v1::detail::less<  
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>>::type is_heap(ExPolicy &&policy, RandIter  
first, RandIter 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<()`). Executed according to the policy.

`comp` has to induce a strict weak ordering on the values.

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.

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.
- **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.

Returns 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 RandIter, typename Comp = hpx::parallel::v1::detail::less>
bool is_heap(RandIter first, RandIter 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: Linear in the distance between *first* and *last*.

Template Parameters

- **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

- **first** – Refers to the beginning of the sequence 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.

Returns The *is_heap* a *bool*. 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 = hpx::parallel::v1::detail::less<  
hpx::parallel::util::detail::algorithm_result<ExPolicy, RandIter>>::type is_heap_until(ExPolicy &&policy,  
RandIter 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<()). Executed according to the policy.

comp has to induce a strict weak ordering on the values.

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.

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.
- **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.

Returns 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 RandIter, typename Comp = hpx::parallel::v1::detail::less>
```

RandIter **is_heap_until**(*RandIter* 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: Linear in the distance between *first* and *last*.

Template Parameters

- **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

- **first** – Refers to the beginning of the sequence 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.

Returns The *is_heap_until* algorithm returns a *RandIter*. 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.

hpx/parallel/algorithms/is_partitioned.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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* = distance(first, last).

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`.

Returns 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<typename ExPolicy, typename FwdIter, typename Pred>
hpx::parallel::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. Executed according to the policy.

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.

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.

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*.

Returns 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.

hpx/parallel/algorithms/is_sorted.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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

```
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*.

Returns 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::detail::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. Executed according to the policy.

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.

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.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

bool pred(const Type &a, const Type &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Returns The *is_sorted* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_sorted* algorithm returns a *bool* if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

```
template<typename FwdIter, typename Pred = hpx::parallel::v1::detail::less>
FwdIter is_sorted_until(FwdIter first, FwdIter last, Pred &&pred = Pred())
```

Returns the first element in the range [first, last) that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most $(N+S-1)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$. $S = \text{number of partitions}$

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*.

Returns 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. Executed according to the policy.

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.

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.

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*.

Returns 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.

hpx/parallel/algorithms/lexicographical_compare.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter1, typename InIter2, typename Pred = hpx::parallel::v1::detail::less>
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 $N1 = \text{std}::\text{distance}(\text{first1}, \text{last})$ and $N2 = \text{std}::\text{distance}(\text{first2}, \text{last2})$.

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*
-

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

Returns 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 ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred =
    hpx::parallel::v1::detail::less>
hpx::parallel::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.

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: Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std}::\text{distance}(\text{first1}, \text{last})$ and $N2 = \text{std}::\text{distance}(\text{first2}, \text{last2})$.

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*
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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

Returns 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.

hpx/parallel/algorithms/make_heap.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename RndIter, typename Comp>
hpx::parallel::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). Executed according to the policy.

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.

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.

Note: Complexity: at most (3*N) comparisons where N = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RndIter** – The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.
- **Comp** – Comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

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

While the signature does not need to have `const &`, the function must not modify the objects passed to it and must be able to accept all values of type (possibly *const*) *Type1* and *Type2* regardless of value category (thus, *Type1 &* is not allowed, nor is *Type1* unless for *Type1* a move is equivalent to a copy. The types *Type1* and *Type2* must be such that an object of type *RandomIt* can be dereferenced and then implicitly converted to both of them.

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.
- **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*.

Returns 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. Executed according to the policy.

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.

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.

Note: Complexity: at most (3*N) comparisons where $N = \text{distance(first, last)}$.

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.
- **RndIter** – The type of the source iterators used for algorithm. This iterator must meet the requirements for 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 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.

Returns 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 RndIter, typename Comp>
void make_heap(RndIter first, RndIter last, Comp &&comp)
```

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3^*N) comparisons where $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- **RndIter** – The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.
- **Comp** – Comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

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

While the signature does not need to have `const &`, the function must not modify the objects passed to it and must be able to accept all values of type (possibly *const*) *Type1* and *Type2* regardless of value category (thus, *Type1 &* is not allowed, nor is *Type1* unless for *Type1* a move is equivalent to a copy. The types *Type1* and *Type2* must be such that an object of type *RandomIt* can be dereferenced and then implicitly converted to both of them.

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*.

Returns The *make_heap* algorithm returns a *void*.

```
template<typename RndIter>
void make_heap(RndIter first, RndIter last)
Constructs a max heap in the range [first, last).
```

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters *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.

Returns The *make_heap* algorithm returns a *void*.

hpx/parallel/algorithms/merge.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename RandIter1, typename RandIter2, typename RandIter3,
typename Comp = hpx::parallel::v1::detail::less
hpx::parallel::util::detail::algorithm_result<ExPolicy, RandIter3>::type merge(ExPolicy &&policy, RandIter1
first1, RandIter1 last1,
RandIter2 first2, RandIter2
last2, RandIter3 dest, Comp
&&comp = Comp())
```

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges. Executed according to the policy.

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.

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.

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.
- **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*.

Returns 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 RandIter1, typename RandIter2, typename RandIter3, typename Comp = hpx::parallel::v1::detail::less>
RandIter3 merge(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.

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

- **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

- **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*

Returns The *merge* algorithm returns a *RandIter3*. The *merge* algorithm returns the destination iterator to the end of the *dest* range.

```
template<typename ExPolicy, typename RandIter, typename Comp = hpx::parallel::v1::detail::less<  
hpx::parallel::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. Executed according to the policy.

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.

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.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

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*

Returns 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 *in-place_merge* algorithm returns the source iterator *last*.

```
template<typename RandIter, typename Comp = hpx::parallel::v1::detail::less>
void inplace_merge(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.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **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

- **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*

Returns The *inplace_merge* algorithm returns a *void*. The *inplace_merge* algorithm returns the source iterator *last*.

hpx/parallel/algorithms/minmax.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter, typename F = hpx::parallel::v1::detail::less>
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* = std::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`.

Returns 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 F = hpx::parallel::v1::detail::less>
hpx::parallel::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*. Executed according to the policy.

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.

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.

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 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`.

Returns 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 = hpx::parallel::v1::detail::less>
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 $N = \text{std::distance}(\text{first}, \text{last})$.

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. 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`.

Returns 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 = hpx::parallel::v1::detail::less>
hpx::parallel::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*. Executed according to the policy.

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.

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.

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 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 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*.

Returns 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 = hpx::parallel::v1::detail::less>
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`.

Returns 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 = hpx::parallel::v1::detail::less>
hpx::parallel::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.

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.

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*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence 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*.

Returns The *minmax_element* algorithm returns a `hpx::future<minmax_element_result<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.

hpx/parallel/algorithms/mismatch.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch(ExPolicy
    &&policy,
    FwdIter1
    first1,
    FwdIter1
    last1,
    FwdIter2
    first2,
    FwdIter2
    last2,
    Pred
    &&op)
```

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2, last2). If last2 is not provided, it denotes first2 + (last1 - first1). Executed according to the policy.

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.

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: Complexity: At most $\min(\text{last1} - \text{first1}, \text{last2} - \text{first2})$ applications of the predicate *op* or *operator==*. 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 *op* or *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator *i* in the range [first1, last1), $*i$ mismatch $*(\text{first2} + (\text{i} - \text{first1}))$. This overload of mismatch uses *operator==* to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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

Returns 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. If no mismatches are found when the comparison reaches `last1` or `last2`, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch(ExPolicy
    &&policy,
    FwdIter1
    first1,
    FwdIter1
    last1,
    FwdIter2
    first2,
    FwdIter2
    last2)
```

Returns the first mismatching pair of elements from two ranges: one defined by $[first1, last1)$ and another defined by $[first2, last2)$. If `last2` is not provided, it denotes $first2 + (last1 - first1)$. Executed according to the policy.

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.

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: Complexity: At most $\min(last1 - first1, last2 - first2)$ applications of `operator==`. If `FwdIter1` and `FwdIter2` meet the requirements of `RandomAccessIterator` and $(last1 - first1) \neq (last2 - first2)$ then no

applications of *operator==* are made.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

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.

Returns 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. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch(ExPolicy
    &&policy,
    FwdIter1
    first1,
    FwdIter1
    last1,
    FwdIter2
    first2,
    Pred
    &&op)
```

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2, last2). If last2 is not provided, it denotes first2 + (last1 - first1). Executed according to the policy.

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.

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: Complexity: At most $\text{last1} - \text{first1}$ applications of the predicate *op* or *operator==*. 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 *op* or *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator *i* in the range $[\text{first1}, \text{last1}]$, $*\text{i}$ mismatch $(*(\text{first2} + (\text{i} - \text{first1}))$. This overload of *mismatch* uses *operator==* to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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

Returns 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. If no mismatches are found when the comparison

reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch(ExPolicy
    &&policy,
    FwdIter1
    first1,
    FwdIter1
    last1,
    FwdIter2
    first2)
```

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2, last2). If last2 is not provided, it denotes first2 + (last1 - first1). Executed according to the policy.

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.

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: Complexity: At most last1 - first1 applications of *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1, last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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. If no mismatches are found when the comparison reaches `last1` or `last2`, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

```
template<typename FwdIter1, typename FwdIter2, typename Pred>
std::pair<FwdIter1, FwdIter2> mismatch(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2,
                                         Pred &&op)
```

Returns the first mismatching pair of elements from two ranges: one defined by $[first1, last1]$ and another defined by $[first2, last2]$. If `last2` is not provided, it denotes $first2 + (last1 - first1)$.

Note: Complexity: At most $\min(last1 - first1, last2 - first2)$ applications of the predicate `op` or `operator==`. If `FwdIter1` and `FwdIter2` meet the requirements of `RandomAccessIterator` and $(last1 - first1) \neq (last2 - first2)$ then no applications of the predicate `op` or `operator==` are made.

Note: The two ranges are considered mismatch if, for every iterator `i` in the range $[first1, last1]$, $*i$ mismatch $*(first2 + (i - first1))$. This overload of `mismatch` uses `operator==` to determine if two elements are mismatch.

Template Parameters

- **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

- **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

Returns The `mismatch` algorithm returns a `std::pair<FwdIter1,FwdIter2>`. If no mismatches are found when the comparison reaches `last1` or `last2`, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

```
template<typename FwdIter1, typename FwdIter2>
std::pair<FwdIter1, FwdIter2> mismatch(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2)
```

Returns the first mismatching pair of elements from two ranges: one defined by $[first1, last1]$ and another defined by $[first2, last2]$. If `last2` is not provided, it denotes $first2 + (last1 - first1)$.

Note: Complexity: At most $\min(last1 - first1, last2 - first2)$ applications of `operator==`. If `FwdIter1` and `FwdIter2` meet the requirements of `RandomAccessIterator` and $(last1 - first1) \neq (last2 - first2)$ then no applications of `operator==` are made.

Note: The two ranges are considered mismatch if, for every iterator `i` in the range $[first1, last1]$, $*i$ mismatch $*(first2 + (i - first1))$. This overload of `mismatch` uses `operator==` to determine if two elements are mismatch.

Template Parameters

- **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.

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.

Returns The `mismatch` algorithm returns a `std::pair<FwdIter1,FwdIter2>`. If no mismatches are found when the comparison reaches `last1` or `last2`, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

```
template<typename FwdIter1, typename FwdIter2, typename Pred>
std::pair<FwdIter1, FwdIter2> mismatch(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, Pred &&op)
```

Returns the first mismatching pair of elements from two ranges: one defined by $[first1, last1]$ and another defined by $[first2, last2]$. If `last2` is not provided, it denotes $first2 + (last1 - first1)$.

Note: Complexity: At most $\text{last1} - \text{first1}$ applications of the predicate *op* or *operator==*. 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 *op* or *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator *i* in the range $[\text{first1}, \text{last1}]$, $*i$ mismatch $*(\text{first2} + (\text{i} - \text{first1}))$. This overload of mismatch uses *operator==* to determine if two elements are mismatch.

Template Parameters

- **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

- **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

Returns The *mismatch* algorithm returns a `std::pair<FwdIter1, FwdIter2>`. If no mismatches are found when the comparison reaches *last1* or *last2*, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

```
template<typename FwdIter1, typename FwdIter2>
std::pair<FwdIter1, FwdIter2> mismatch(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2)
```

Returns the first mismatching pair of elements from two ranges: one defined by $[\text{first1}, \text{last1}]$ and another defined by $[\text{first2}, \text{last2}]$. If *last2* is not provided, it denotes $\text{first2} + (\text{last1} - \text{first1})$.

Note: Complexity: At most $\text{last1} - \text{first1}$ applications of *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and $(\text{last1} - \text{first1}) \neq (\text{last2} - \text{first2})$ then no applications of *operator==* are made.

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.

Template Parameters

- **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

- **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.

Returns The *mismatch* algorithm returns a *std::pair<FwdIter1, FwdIter2>*. If no mismatches are found when the comparison reaches *last1* or *last2*, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

hpx/parallel/algorithms/move.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
hpx::parallel::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. Executed according to the policy.

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.

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.

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.

Returns 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.

```
template<typename FwdIter1, typename FwdIter2>
FwdIter2 move(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.

Note: Complexity: Performs exactly *last - first* move assignments.

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 (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.
- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns a `FwdIter2`. The *move* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

hpx/parallel/algorithms/nth_element.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename RandomIt, typename Pred = hpx::parallel::v1::detail::less>
void nth_element(RandomIt first, RandomIt nth, RandomIt last, Pred &&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. Executed according to the policy.

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<>`.

Returns The `nth_element` algorithms returns nothing.

```
template<typename ExPolicy, typename RandomIt, typename Pred = hpx::parallel::v1::detail::less>
```

```
void nth_element(ExPolicy &&policy, RandomIt first, RandomIt nth, RandomIt last, Pred &&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.

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.

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<>.

Returns The *nth_element* algorithms returns nothing.

hpx/parallel/algorithms/partial_sort.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename RandIter, typename Comp = hpx::parallel::v1::detail::less>
RandIter 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.

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.

Returns The *partial_sort* algorithm returns a *RandIter* that refers to *last*.

```
template<typename ExPolicy, typename RandIter, typename Comp = hpx::parallel::v1::detail::less>
parallel::util::detail::algorithm_result_t<ExPolicy, RandIter> partial_sort(ExPolicy &&policy, RandIter
first, RandIter middle, RandIter
last, Comp &&comp =
Comp())
```

Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

Note: Complexity: Approximately (last - first) * log(middle - first) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns The *partial_sort* 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 iterator returned refers to *last*.

hpx/parallel/algorithms/partial_sort_copy.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter, typename RandIter, typename Comp = hpx::parallel::v1::detail::less>
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.

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.

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 an 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`.

Returns The `partial_sort_copy` algorithm returns a *RandomIt*. The algorithm returns an iterator to the element defining the upper boundary of the sorted range i.e. $d_{\text{first}} + \min(last - first, d_{\text{last}} - d_{\text{first}})$

```
template<typename ExPolicy, typename FwdIter, typename RandIter, typename Comp =
hpx::parallel::v1::detail::less>
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 $[d_{\text{first}}, d_{\text{last}}]$. At most $d_{\text{last}} - d_{\text{first}}$ of the elements are placed sorted to the range $[d_{\text{first}}, d_{\text{first}} + n]$ where n is the number of elements to sort ($n = \min(last - first, d_{\text{last}} - d_{\text{first}})$). Executed according to the policy.

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.

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

- **ExPolicy** – The type of the execution policy to use (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.

Returns The *partial_sort_copy* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator to the element defining the upper boundary of the sorted range i.e. *d_first + min(last - first, d_last - d_first)*

hpx/parallel/algorithms/partition.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter, typename Pred, typename Proj = parallel::util::projection_identity>
FwdIter partition(FwdIter first, FwdIter last, Pred &&pred, Proj &&proj = Proj())
```

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked 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 a 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 *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.

Returns The *partition* algorithm 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 =  
parallel::util::projection_identity>  
parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter> partition(ExPolicy &&policy, FwdIter first,  
FwdIter last, Pred &&pred, Proj  
&&proj = Proj())
```

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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.
- **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 *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.

Returns 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 = parallel::util::projection_identity>
BidirIter stable_partition(BidirIter first, BidirIter last, F &&f, Proj &&proj = Proj())
```

Permutes the elements in the range [first, last) such that there exists an iterator *i* such that for every iterator *j* in the range [first, *i*) *INVOKE(f, INVOKE(proj, *j)) != false*, and for every iterator *k* in the range [*i*, last), *INVOKE(f, INVOKE(proj, *k)) == false*

The invocations of *f* in the parallel *stable_partition* algorithm invoked 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 a 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

- **first** – Refers to the beginning of the sequence 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.

Returns The *stable_partition* algorithm returns an iterator *i* such that for every iterator *j* in the range $[\text{first}, i)$, $f(*j) \neq \text{false}$ $\text{INVOKE}(f, \text{INVOKED}(\text{proj}, *j)) \neq \text{false}$, and for every iterator *k* in the range $[i, \text{last})$, $f(*k) == \text{false}$ $\text{INVOKED}(f, \text{INVOKED}(\text{proj}, *k)) == \text{false}$. The relative order of the elements in both groups is preserved.

```
template<typename ExPolicy, typename BidirIter, typename F, typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result_t<ExPolicy, BidirIter> stable_partition(ExPolicy &&policy,
                                                               BidirIter first, BidirIter
                                                               last, F &&f, Proj &&proj
                                                               = Proj())
```

Permutes the elements in the range $[\text{first}, \text{last})$ such that there exists an iterator *i* such that for every iterator *j* in the range $[\text{first}, i)$ $\text{INVOKED}(f, \text{INVOKED}(\text{proj}, *j)) \neq \text{false}$, and for every iterator *k* in the range $[i, \text{last})$, $\text{INVOKED}(f, \text{INVOKED}(\text{proj}, *k)) == \text{false}$

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.

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.

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 a 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.
- **first** – Refers to the beginning of the sequence 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.

Returns The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range $[\text{first}, i)$, $f(*j) != \text{false}$ $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *j)) != \text{false}$, and for every iterator k in the range $[i, \text{last})$, $f(*k) == \text{false}$ $\text{INVOKE}(f, \text{INVOKE}(\text{proj}, *k)) == \text{false}$. The relative order of the elements in both groups is preserved. 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 = parallel::util::projection_identity>
std::pair<FwdIter2, FwdIter3> partition_copy(FwdIter1 first, FwdIter1 last, FwdIter2 dest_true, FwdIter3 dest_false, Pred &&pred, Proj &&proj = Proj())
```

Copies the elements in the range, defined by $[\text{first}, \text{last})$, to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *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 *pred*.

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.

Returns 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 **Proj** = *parallel::util::projection_identity*>

```
parallel::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  
    =  
    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.

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.

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.
- **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.

Returns 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.

hpx/parallel/algorithms/reduce.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter, typename F, typename T = typename
std::iterator_traits<FwdIter>::value_type>
hpx::parallel::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)). Executed according to the policy.

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.

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.

Note: Complexity: $O(\text{last} - \text{first})$ applications of the predicate f .

Note: GENERALIZED_SUM($\text{op}, \text{a}_1, \dots, \text{a}_N$) is defined as follows:

- a_1 when N is 1
 - $\text{op}(\text{GENERALIZED_SUM}(\text{op}, \text{b}_1, \dots, \text{b}_K), \text{GENERALIZED_SUM}(\text{op}, \text{b}_M, \dots, \text{b}_N))$, where:
 - $\text{b}_1, \dots, \text{b}_N$ may be any permutation of $\text{a}_1, \dots, \text{a}_N$ and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source begin and end 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 *reduce* 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.
- **init** – The initial value for the generalized sum.
- **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.

Returns 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).

```
template<typename ExPolicy, typename FwdIter, typename T = typename  
std::iterator_traits<FwdIter>::value_type>
```

`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)). Executed according to the policy.

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.

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.

Note: Complexity: O(*last - first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
 - op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - 1 < K+1 = M <= N.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 a 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.

Returns 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).

template<typename **ExPolicy**, typename **FwdIter**>

```
hpx::parallel::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)). Executed according to the policy.

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.

The reduce operations in the parallel *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 *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last - first*) applications of the operator+().

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
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 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.

Returns 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).

template<typename **FwdIter**, typename **F**, typename **T** = typename `std::iterator_traits<FwdIter>::value_type`>
T **reduce**(`FwdIter` first, `FwdIter` last, `T` init, `F` &&f)

Returns GENERALIZED_SUM(f, init, *first, ..., *(first + (last - first) - 1)). Executed according to the policy.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: $O(last - first)$ applications of the predicate `f`.

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 Parameters

- **FwdIter** – The type of the source begin and end 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 *reduce* requires `F` to meet the requirements of *CopyConstructible*.
- **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.
- **init** – The initial value for the generalized sum.
- **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 `InIter` can be dereferenced and then implicitly converted to any of those types.

Returns The *reduce* algorithm returns `T`. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

```
template<typename FwdIter, typename T = typename std::iterator_traits<FwdIter>::value_type>
T reduce(FwdIter first, FwdIter last, T init)
```

Returns GENERALIZED_SUM(+, init, *first, ..., *(first + (last - first) - 1)). Executed according to the policy.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last - first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
 - op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - 1 < K+1 = M <= N.
-

Template Parameters

- **FwdIter** – The type of the source begin and end iterators used (deduced). This iterator 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

- **first** – Refers to the beginning of the sequence 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.

Returns The *reduce* algorithm returns a *T*. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

```
template<typename FwdIter>
std::iterator_traits<FwdIter>::value_type reduce(FwdIter first, FwdIter last)
```

Returns GENERALIZED_SUM(+, T(), *first, ..., *(first + (last - first) - 1)). Executed according to the policy.

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last - first*) applications of the operator+().

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
 - $1 < K+1 = M \leq N$.
-

Template Parameters `FwdIter` – The type of the source begin and end iterators used (deduced).

This iterator type must meet the requirements of an input 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.

Returns The *reduce* algorithm returns *T* (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].

hpx/parallel/algorithms/reduce_by_key.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

Functions

```
template<typename ExPolicy, typename RanIter, typename RanIter2, typename FwdIter1,
         typename FwdIter2, typename Compare = std::equal_to<typename
std::iterator_traits<RanIter>::value_type>, typename Func = std::plus<typename
std::iterator_traits<RanIter2>::value_type>>
```

```
util::detail::algorithm_result<ExPolicy, util::in_out_result<FwdIter1, FwdIter2>>::type reduce_by_key(ExPolicy  
    &&policy,  
    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.

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.

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 a forward iterator.

- **FwdIter2** – The type of the iterator representing the destination value range (deduced). This iterator type must meet the requirements of a 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 *reduce_by_key* requires *Func* 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.

Returns 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.

hpx/parallel/algorithms/remove.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter, typename T = typename std::iterator_traits<FwdIter>::value_type>
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

- **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.

Returns 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 = typename
std::iterator_traits<FwdIter>::value_type>
parallel::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*. Executed according to the policy.

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.

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.

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.

Returns 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:

<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`.

Returns 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>
parallel::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. Executed according to the policy.

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.

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.

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`.

Returns 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.

hpx/parallel/algorithms/remove_copy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter, typename OutIter, typename T = typename
std::iterator_traits<InIter>::value_type>
OutIter 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 `pred`, here comparison operator.

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.
- **value** – Value to be removed.

Returns 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 = typename
std::iterator_traits<InIter>::value_type>
```

```
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type remove_copy(ExPolicy &&policy,
    FwdIter1 first, FwdIter1 last,
    FwdIter2 dest, T const
    &value)
```

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the comparison operator returns false when compare to *value*. The order of the elements that are not removed is preserved. Executed according to the policy.

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.

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.

Note: Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *pred*, here comparison 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.
- **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.
- **value** – Value to be removed.

Returns 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>
OutIter 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 *pred* 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 `pred`.

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).

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`.

Returns 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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 `pred` returns `false`. The order of the elements that are not removed is preserved. Executed according to the policy.

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.

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.

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.
- **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*.

Returns 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.

hpx/parallel/algorithms/replace.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter, typename T = typename std::iterator_traits<InIter>::value_type>
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.

Returns The *replace* algorithm returns a *void*.

```
template<typename ExPolicy, typename FwdIter, typename T = typename std::iterator_traits<FwdIter>::value_type>
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). Executed according to the policy.

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.

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.

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.

Returns 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 = typename std::iterator_traits<Iter>::value_type>
void replace_if(Iter first, Iter last, Pred &&pred, T const &new_value)
```

Replaces all elements satisfying specific criteria (for which predicate *pred* 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 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.

Returns The `replace_if` algorithm returns `void`.

```
template<typename ExPolicy, typename FwdIter, typename Pred, typename T = typename  

std::iterator_traits<FwdIter>::value_type>  

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). Executed according to the policy.

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.

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.

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 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.

Returns 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 = typename  
std::iterator_traits<OutIter>::value_type>  
OutIter replace_copy(InIter first, InIter last, OutIter dest, T const &old_value, T const &new_value)
```

Copies 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.

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.

Returns 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 = typename
std::iterator_traits<FwdIter2>::value_type>
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. Executed according to the policy.

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.

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.

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.

Returns 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 = typename
std::iterator_traits<OutIter>::value_type>
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 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.

Returns 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 =
typename std::iterator_traits<FwdIter2>::value_type>
```

```
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: $\text{(INVOKE}(f, *(\text{first} + (\text{i} - \text{result}))) \neq \text{false}$

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.

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.

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 *replace_copy_if* 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.
- **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:

<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 `FwdIter1` can be dereferenced and then implicitly converted to `Type`.

- **new_value** – Refers to the new value to use as the replacement.

Returns 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.

hpx/parallel/algorithms/reverse.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 a 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.

Returns The `reverse` algorithm returns `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 $i < (last-first)/2$. Executed according to the policy.

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.

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.

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 a 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.

Returns 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 in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(\text{dest} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [first, last) and [dest, dest+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a 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.

Returns 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type reverse_copy(ExPolicy &&policy,
BidirIter first, BidirIter last,
FwdIter dest)
```

Copies the elements from the range [first, last) to another range beginning at dest in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(\text{dest} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [first, last) and [dest, dest+(last-first)) respectively) overlap, the behavior is undefined. Executed according to the policy.

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.

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.

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 a bidirectional iterator.
- **FwdIter** – 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 begin of the destination range.

Returns 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.

hpx/parallel/algorithms/rotate.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

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*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Template Parameters **FwdIter** – The type of the source iterators used (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.
- **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.

Returns 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<typename ExPolicy, typename FwdIter>
parallel::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. Executed according to the policy.

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.

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: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *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 executes the assignments.
- **FwdIter** – The type of the source iterators used (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.
- **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.

Returns 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 a forward iterator.
- **OutIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a 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.

Returns 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>
parallel::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. Executed according to the policy.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

The assignments in the parallel *rotate_copy* algorithm execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each 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 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.

Parameters

- **policy** – The execution policy 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.

Returns 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.

hpx/parallel/algorithms/search.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter, typename FwdIter2, typename Pred = parallel::v1::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 ($S \cdot N$) 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 a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *search* 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

```
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

Returns 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 =
parallel::v1::detail::equal_to
parallel::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. Executed according to the policy.

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.

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.

Note: Complexity: at most (*S***N*) comparisons where *S* = distance(*s_first*, *s_last*) and *N* = distance(*first*, *last*).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *search* 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

Returns 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 = parallel::v1::detail::equal_to>
FwdIter search_n(FwdIter first, std::size_t count, FwdIter2 s_first, FwdIter2 s_last, Pred &&op = Pred())
```

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_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 a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of `search_n` 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

Returns 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 =  
parallel::v1::detail::equal_to>  
parallel::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. Executed according to the policy.

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.

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.

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 a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *search_n* 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

Returns 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 $[s_{\text{first}}, s_{\text{last}})$ in range $[first, first+count]$. If the length of the subsequence $[s_{\text{first}}, s_{\text{last}})$ 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.

hpx/parallel/algorithms/set_difference.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = hpx::parallel::v1::detail::less>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter3>::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 `pred`. Executed according to the policy.

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.

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`).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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 a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

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

The signature does not need to have `const &`, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_difference* algorithm returns a `hpx::future<FwdIter3>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred =
hpx::parallel::v1::detail::less>
```

```
FwdIter3 set_difference(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 *pred*.

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.

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.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **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 *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_difference* algorithm returns a *FwdIter3*. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

`hpx/parallel/algorithms/set_intersection.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace `hpx`

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename  

Pred = hpx::parallel::v1::detail::less>  

hpx::parallel::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 *pred*. Executed according to the policy.

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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 a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.

- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator with sequential execution.
- **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 *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_intersection* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = hpx::parallel::v1::detail::less>
FwdIter3 set_intersection(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 *pred*.

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.

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.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator with sequential execution.
- **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

- **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 *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_intersection* algorithm returns a *FwdIter3*. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx/parallel/algorithms/set_symmetric_difference.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename  
Pred = hpx::parallel::v1::detail::less>  
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter3>::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 *pred*. Executed according to the policy.

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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 a forward iterator.

- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.
- **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 *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_symmetric_difference* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred =  
    hpx::parallel::v1::detail::less>  
FwdIter3 set_symmetric_difference(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 *pred*.

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.

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.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.
- **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

- **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 *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The `set_symmetric_difference` algorithm returns a *FwdIter3*. The `set_symmetric_difference` algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx/parallel/algorithms/set_union.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename  

Pred = hpx::parallel::v1::detail::less>  

hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter3>::type set_union(ExPolicy &&policy,  

FwdIter1 first1, FwdIter1 last1, FwdIter2 first2,  

FwdIter2 last2, FwdIter3  

dest, Pred &&op =  

Pred())
```

Constructs a sorted range beginning at *dest* consisting of all elements present in one or both sorted ranges [*first1*, *last1*] and [*first2*, *last2*]. This algorithm expects both input ranges to be sorted with the given binary predicate *pred*. Executed according to the policy.

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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 a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.

- **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 *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_union* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = hpx::parallel::v1::detail::less>
FwdIter3 set_union(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 *pred*. Executed according to the policy.

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.

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.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.

- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.
- **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

- **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 *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_union* algorithm returns a *FwdIter3*. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx/parallel/algorithms/shift_left.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

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 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.

Returns 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>
hpx::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. Executed according to the policy.

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.

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: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 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.

Returns 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.

hpx/parallel/algorithms/shift_right.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

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 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.

Returns 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>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter> shift_right(ExPolicy &&policy, FwdIter first, FwdIter last, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- n + i. Executed according to the policy.

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.

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: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 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.

Returns 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.

hpx/parallel/algorithms/sort.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename RandomIt, typename Comp = hpx::parallel::v1::detail::less, typename Proj = parallel::util::projection_identity>
void sort(RandomIt first, RandomIt last, Comp &&comp, Proj &&proj = Proj())
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i + n*)), INVOKE(*proj*, **i*)) == false.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

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 `hpx::parallel::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.

Returns The *sort* algorithm returns *void*.

```
template<typename ExPolicy, typename RandomIt, typename Comp = hpx::parallel::v1::detail::less, typename Proj = parallel::util::projection_identity>
```

`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<()). Executed according to the policy.

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`.

comp has to induce a strict weak ordering on the values.

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.

Note: Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ 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 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 `hpx::parallel::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.

Returns The `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.

hpx/parallel/algorithms/sort_by_key.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

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,
ValueIter
value_first,
Compare
&&comp =
Compare()
```

Sorts one range of data using keys supplied in another range. The key elements in the range [key_first, key_last) are sorted in ascending order with the corresponding elements in the value range moved to follow the sorted order. The algorithm is not stable, the order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()). Executed according to the policy.

A sequence is sorted with respect to a comparator *comp* 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*, *(*i + n*), **i*) == false.

comp has to induce a strict weak ordering on the values.

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.

Note: Complexity: O(Nlog(N)), where N = std::distance(first, 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 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.
- **Compare** – 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.

Returns 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.

hpx/parallel/algorithms/stable_sort.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename RandomIt, typename Comp = hpx::parallel::v1::detail::less, typename Proj = parallel::util::projection_identity>
void stable_sort(RandomIt first, RandomIt last, Comp &&comp = Comp(), Proj &&proj = Proj())
Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).
```

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i* + *n* is a valid iterator pointing to an element of the sequence, and `INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false`.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *stable_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ comparisons.

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 `parallel::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.

Returns The `stable_sort` algorithm returns `void`.

```
template<typename ExPolicy, typename RandomIt, typename Comp = hpx::parallel::v1::detail::less,
typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy>::type stable_sort(ExPolicy &&policy, RandomIt first,
RandomIt last, Comp &&comp =
Comp(), Proj &&proj = Proj())
```

Sorts the elements in the range $[first, last)$ in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object `comp` (defaults to using `operator<()`). Executed according to the policy.

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`.

`comp` has to induce a strict weak ordering on the values.

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.

Note: Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ 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 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 `parallel::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.

Returns 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 `void` otherwise.

[hpx/parallel/algorithms/starts_with.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter1, typename InIter2, typename Pred = hpx::parallel::v1::detail::equal_to,
         typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
bool starts_with(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Pred &&pred = Pred(), Proj1
                  &&proj1 = Proj1(), Proj2 &&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 $\min(N1, N2)$ applications of the predicate and both projections.

Template Parameters

- **InIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the destination iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Pred** – The binary predicate that compares the projected elements. This defaults to `hpx::parallel::v1::detail::equal_to`.
- **Proj1** – The type of an optional projection function for the source range. This defaults to `hpx::parallel::util::projection_identity`.
- **Proj2** – The type of an optional projection function for the destination range. This defaults to `hpx::parallel::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 *pred* 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 *pred* is invoked.

Returns 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 InIter1, typename InIter2, typename Pred =
hpx::parallel::v1::detail::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 =
parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, bool>::type starts_with(ExPolicy &&policy, InIter1 first1,
InIter1 last1, InIter2 first2,
InIter2 last2, Pred &&pred =
Pred(), Proj1 &&proj1 = Proj1(),
Proj2 &&proj2 = Proj2())
```

Checks whether the second range defined by $[first1, last1)$ matches the prefix of the first range defined by $[first2, last2)$. Executed according to the policy.

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

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **InIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the destination iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Pred** – The binary predicate that compares the projected elements. This defaults to `hpx::parallel::v1::detail::equal_to`.
- **Proj1** – The type of an optional projection function for the source range. This defaults to `hpx::parallel::util::projection_identity`.
- **Proj2** – The type of an optional projection function for the destination range. This defaults to `hpx::parallel::util::projection_identity`.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the two ranges projected by *proj1* and *proj2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *pred* 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 *pred* is invoked.

Returns 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.

[hpx/parallel/algorithms/swap_ranges.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 a forward iterator.
- **FwdIter2** – The type of the second range of iterators to swap (deduced). This iterator type must meet the requirements of a 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.

Returns 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*. Executed according to the policy.

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.

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.

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 a forward iterator.
- **FwdIter2** – The type of the second range of iterators to swap (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.
- **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.

Returns 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*.

hpx/parallel/algorithms/transform.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter1, typename FwdIter2, typename F>
FwdIter2 transform(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.

Note: Complexity: Exactly *last - first* applications of *f*

Template Parameters

- **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.
- **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

- **first** – Refers to the beginning of the sequence 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`.

Returns The `transform` algorithm returns a `FwdIter2`. 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 F>
parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> 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. Executed according to the policy.

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.

The invocations of `f` in the parallel `transform` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: 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 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.
- **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*.

Returns The *transform* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *parallel_task_policy* and returns *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 FwdIter1, typename FwdIter2, typename FwdIter3, typename F>
FwdIter3 transform(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.

Note: Complexity: Exactly *last - first* applications of *f*

Template Parameters

- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators for the second range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination 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 *transform* requires *F* to meet the requirements of *CopyConstructible*.

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.
- **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`.

Returns The `transform` algorithm returns a `FwdIter3`. The `transform` algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename F>
parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter3> transform(ExPolicy &&policy, 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 $dest$. Executed according to the policy.

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.

The invocations of f in the parallel `transform` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: 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 a forward iterator.
- **FwdIter2** – The type of the source iterators for the second range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination 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 `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*.

Returns The *transform* algorithm returns a `hpx::future<FwdIter3>` if the execution policy is of type `parallel_task_policy` and returns *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.

hpx/parallel/algorithms/transform_exclusive_scan.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename InIter, typename OutIter, typename BinOp, typename UnOp, typename T = typename std::iterator_traits<InIter>::value_type>
OutIter transform_exclusive_scan(InIter first, InIter last, OutIter dest, T init, BinOp &&binary_op,
                                UnOp &&unary_op)
```

Transforms each element in the range [first, last) with *unary_op*, then computes an exclusive prefix sum operation using *binary_op* over the resulting range, with *init* as the initial value, and writes the results to the range beginning at *dest*. “exclusive” means that the i-th input element is not included in the i-th sum. Formally, assigns through each iterator *i* in [dest, *d*_first + (last - first)) the value of the generalized noncommutative sum of *init*, *unary_op*(*j)... for every *j* in [first, first + (i - d)_first)) over *binary_op*, where generalized noncommutative sum GNSUM(*op*, *a*1, ..., *a*N) is defined as follows:

- if N=1, *a*1
- if N > 1, *op*(GNSUM(*op*, *a*1, ..., *a*K), GNSUM(*op*, *a*M, ..., *a*N)) for any K where 1 < K+1 = M <= N In other words, the summation operations may be performed in arbitrary order, and the behavior is nondeterministic if *binary_op* is not associative.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *unary_op* nor *binary_op* shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

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 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.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.
- **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.
- **binary_op** – Binary *FunctionObject* that will be applied to the result of *unary_op*, the results of other *binary_op*, and *init*.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename BinOp, typename UnOp,  
typename T = typename std::iterator_traits<FwdIter1>::value_type>
```

```
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($binary_op$, $init$, $conv(*first), \dots, conv(*(\text{first} + (i - result) - 1))$). Executed according to the policy.

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.

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 *unary_op* nor *binary_op* shall invalidate iterators or subranges, or modify elements in the ranges $[first, last)$ or $[result, result + (last - first))$.

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of each of *binary_op* and *unary_op*.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.

- **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.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init*.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns 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.

hpx/parallel/algorithms/transform_inclusive_scan.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

Neither *binary_op* nor *unary_op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

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.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

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 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.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.

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.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns 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.

```
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
    &&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))))). Executed according to the policy.

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.

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 *binary_op* nor *unary_op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The difference between *inclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

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.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence 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.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns 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.

```
template<typename InIter, typename OutIter, typename BinOp, typename UnOp, typename T = typename  
std::iterator_traits<InIter>::value_type>  
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), ..., 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.

Neither *binary_op* nor *unary_op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

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. If *binary_op* is not mathematically associative, the behavior of *transform_inclusive_scan* may be non-deterministic.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) is defined as:

- a1 when N is 1
 - op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN)) where 1 < K+1 = M <= N.
-

Template 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.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.
- **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.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.
- **init** – The initial value for the generalized sum.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename BinOp, typename UnOp,  
typename T = typename std::iterator_traits<FwdIter1>::value_type>  
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type transform_inclusive_scan(ExPolicy  
    &&policy,  
    FwdIter1  
    first,  
    FwdIter1  
    last,  
    FwdIter2  
    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(op, init, conv(*first), ..., conv(*(first + (i - result)))). Executed according to the policy.

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.

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 `binary_op` nor `unary_op` shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

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. If `binary_op` is not mathematically associative, the behavior of `transform_inclusive_scan` may be non-deterministic.

Note: Complexity: O(last - first) applications of each of `binary_op` and `unary_op`.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) is defined as:

- a1 when N is 1
 - op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN)) where 1 < K+1 = M <= N.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.
- **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.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.
- **init** – The initial value for the generalized sum.

Returns 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.

hpx/parallel/algorithms/transform_reduce.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename FwdIter, typename T, typename Reduce, typename Convert>
hpx::parallel::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))). Executed according to the policy.

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.

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.

Note: Complexity: $O(\text{last} - \text{first})$ applications of the predicates *red_op* and *conv_op*.

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, ..., *b*_N may be any permutation of *a*₁, ..., *a*_N and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 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.
- **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 *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*.

Returns 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).

template<typename *InIter*, typename *T*, typename *Reduce*, typename *Convert*>
T **transform_reduce**(*InIter* first, *InIter* 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 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.

Note: Complexity: O(*last - first*) applications of the predicates *red_op* and *conv_op*.

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, ..., *b*N may be any permutation of *a*1, ..., *a*N and
 - 1 < K+1 = M <= N.

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 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

- **first** – Refers to the beginning of the sequence 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.

- **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 *InIter* 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*.

Returns The *transform_reduce* algorithm returns a *T*. 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).

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename TExPolicy, 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*. Executed according to the policy.

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.

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.

Note: Complexity: $O(last - first)$ applications each of *reduce* and *transform*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 a forward iterator.
- **FwdIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of a 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.

Returns 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 InIter1, typename InIter2, typename T>
T transform_reduce(InIter1 first1, InIter1 last1, InIter2 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.

Note: Complexity: $O(last - first)$ applications each of *reduce* and *transform*.

Template Parameters

- **InIter1** – The type of the first source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as return) values (deduced).

Parameters

- **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.

Returns The *transform_reduce* algorithm returns a *T*.

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Reduce,
typename Convert>
hpx::parallel::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. Executed according to the policy.

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.

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.

Note: Complexity: $O(\text{last} - \text{first})$ applications each of *reduce* and *transform*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 a forward iterator.
- **FwdIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of a 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 *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have **const&**, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Ret` must be such that it can be implicitly converted to an object for the second argument type of `red_op`.

Returns 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 InIter1, typename InIter2, typename T, typename Reduce, typename Convert>
T transform_reduce(ExPolicy &&policy, InIter1 first1, InIter1 last1, InIter2 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`.

Note: Complexity: $O(last - first)$ applications each of `reduce` and `transform`.

Template Parameters

- **InIter1** – The type of the first source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an input 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

- **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 `conv_op`. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Ret` must be such that it can be implicitly converted to a type of `T`.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Ret` must be such that it can be implicitly converted to an object for the second argument type of `red_op`.

Returns The `transform_reduce` algorithm returns a `T`.

hpx/parallel/algorithms/transform_reduce_binary.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/parallel/algorithms/uninitialized_copy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

Returns 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. Executed according to the policy.

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.

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.

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.

Returns 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

- **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.

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.

Returns The *uninitialized_copy_n* algorithm returns a *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>
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.

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.

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 a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns 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.

[hpx/parallel/algorithms/uninitialized_default_construct.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 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.

Returns 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. Executed according to the policy.

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.

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.

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.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns 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_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.

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.

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.

Returns The *uninitialized_default_construct_n* algorithm returns a *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. Executed according to the policy.

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.

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.

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.

Returns 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.

`hpx/parallel/algorithms/uninitialized_fill.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

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 a 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.

Returns 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. Executed according to the policy.

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.

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.

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 a 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.

Returns The *uninitialized_fill* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

```
template<typename FwdIter, typename Size, typename T>
FwdIter uninitialized_fill(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.

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.

Returns 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. Executed according to the policy.

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.

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.

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.

Returns 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.

[hpx/parallel/algorithms/uninitialized_move.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

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.

Returns 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. Executed according to the policy.

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.

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.

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 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.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence 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.

Returns 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>
std::pair<InIter, FwdIter> uninitialized_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.

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.

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.

Returns The *uninitialized_move_n* algorithm returns a returns `std::pair<InIter,FwdIter>`. The *uninitialized_move_n* algorithm returns A pair whose first element is an iterator to the element past the last element moved in the source range, and whose second element is an iterator to the element past the last element moved in the destination range.

```
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2>
```

```
parallel::util::detail::algorithm_result<ExPolicy, std::pair<FwdIter1, FwdIter2>>::type uninitialized_move_n(ExPolicy  
    &&policy,  
    FwdIter1  
    first,  
    Size  
    count,  
    FwdIter2  
    dest)
```

Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state. Executed according to the policy.

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.

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.

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 a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_move_n* algorithm returns a *hpx::future<std::pair<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1, FwdIter2>* otherwise. The *uninitialized_move_n* algorithm returns A pair whose first element is an iterator to the element past the last element moved in the source range, and whose second element is an iterator to the element past the last element moved in the destination range.

hpx/parallel/algorithms/uninitialized_value_construct.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

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 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.

Returns 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. Executed according to the policy.

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.

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.

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.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns 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.

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.

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.

Returns 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.

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.

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.

Parameters

- **policy** – The execution policy 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.

Returns 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.

hpx/parallel/algorithms/unique.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter, typename Pred = hpx::parallel::v1::detail::equal_to, typename Proj = parallel::util::projection_identity>
FwdIter unique(FwdIter first, FwdIter last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

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 a 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 parallel::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 *pred* is invoked.

Returns 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 = hpx::parallel::v1::detail::equal_to,
         typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type unique(ExPolicy &&policy, FwdIter first,
                                                               FwdIter last, Pred &&pred =
                                                               Pred(), Proj &&proj = Proj())
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range. Executed according to the policy.

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.

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.

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 a 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 parallel::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 *pred* is invoked.

Returns 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, typename Pred = hpx::parallel::v1::detail::equal_to,
         typename Proj = parallel::util::projection_identity>
OutIter unique_copy(InIter first, InIter last, OutIter dest, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

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 parallel::util::projection_identity

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
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 *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 *pred* is invoked.

Returns 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 =  
hpx::parallel::v1::detail::equal_to, typename Proj = parallel::util::projection_identity>  
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type unique_copy(ExPolicy &&policy,  
FwdIter1 first, FwdIter1 last,  
FwdIter2 dest, Pred &&pred  
= Pred(), Proj &&proj =  
Proj())
```

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied. Executed according to the policy.

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.

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.

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 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 *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 `parallel::util::projection_identity`.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *pred* is invoked.

Returns 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.

hpx/parallel/container_algorithms/adjacent_difference.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename FwdIter1, typename FwdIter2, typename Sent>
FwdIter2 adjacent_difference(FwdIter1 first, Sent last, FwdIter2 dest)
```

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

Template Parameters

- **FwdIter1** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – 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.

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.
- **dest** – Refers to the beginning of the destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

```
template<typename Rng, typename FwdIter2>
FwdIter2 adjacent_difference(Rng &&rng, FwdIter2 dest)
```

Searches the *rng* 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

Template Parameters

- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **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 sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> adjacent_difference(ExPolicy
&&policy,
FwdIter1
first, Sent
last,
FwdIter2
dest)
```

Searches the range [first, last) for two consecutive identical elements.

Note: Complexity: Exactly the smaller of $(\text{result} - \text{first}) + 1$ and $(\text{last} - \text{first}) - 1$ application of the predicate where *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.
- **FwdIter1** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – 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.

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 destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

```
template<typename ExPolicy, typename Rng, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> adjacent_difference(ExPolicy
    &&policy,
    Rng
    &&rng,
    FwdIter2
    dest)
```

Searches the *rng* for two consecutive identical elements.

Note: Complexity: Exactly the smaller of $(\text{result} - \text{first}) + 1$ and $(\text{last} - \text{first}) - 1$ application of the predicate where *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.
- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **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.
- **dest** – Refers to the beginning of the destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

```
template<typename FwdIter1, typename Sent, typename FwdIter2, typename Op>
```

`FwdIter2 adjacent_difference(FwdIter1 first, Sent last, FwdIter2 dest, Op &&op)`

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

Template Parameters

- **FwdIter1** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – 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.
- **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

- **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 destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
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 an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename Rng, typename FwdIter2, typename Op>
`FwdIter2 adjacent_difference(Rng &&rng, FwdIter2 dest, Op &&op)`

Searches the *rng* for two consecutive identical elements.

Template Parameters

- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input 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

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
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 an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.?

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **Op**>

```
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> adjacent_difference(ExPolicy
    &&policy,
    FwdIter1
    first, Sent
    last,
    FwdIter2
    dest, Op
    &&op)
```

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

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – 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.
- **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 destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
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 an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.?

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

```
template<typename ExPolicy, typename Rng, typename FwdIter2, typename Op>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> adjacent_difference(ExPolicy
&&policy,
Rng
&&rng,
FwdIter2
dest, Op
&&op)
```

Searches the *rng* 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

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input 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.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
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 an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

[hpx/parallel/container_algorithms/adjacent_find.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename FwdIter, typename Sent, typename Proj = hpx::parallel::util::projection_identity,  
typename Pred = detail::equal_to>  
FwdIter adjacent_find(FwdIter first, Sent last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Searches the range [first, last) for two consecutive identical elements.

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

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.

Returns The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename Proj =  
hpx::parallel::util::projection_identity, typename Pred = detail::equal_to>  
parallel::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.

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*.

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.

Returns 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_find(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

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.

Returns The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is returned.

```
template<typename ExPolicy, typename Rng, typename Proj = hpx::parallel::util::projection_identity,
typename Pred = detail::equal_to>
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type adjacent_
```

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.

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`.

Note: Complexity: Exactly the smaller of `(result - std::begin(rng)) + 1` and `(std::begin(rng) - std::end(rng)) - 1` applications of the predicate where `result` is the value returned

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). 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.

Returns 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.

hpx/parallel/container_algorithms/all_any_none.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Rng, typename F, typename Proj =  
    hpx::parallel::util::projection_identity>  
hpx::parallel::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.

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.

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.

Returns 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 Iter, typename Sent, typename F, typename Proj =
hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type none_of(ExPolicy &&policy, Iter
first, Sent 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.

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.

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.
- **Iter** – The type of the source iterators used for the range (deduced).
- **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 *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 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.
- **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.

Returns 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 Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
bool none_of(Rng &&rng, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for no elements in the range *rng*.

Note: Complexity: At most `std::distance(begin(rng), end(rng))` applications of the predicate *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 *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

- **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 ob-

jects 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.

Returns 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 Iter, typename Sent, typename F, typename Proj =
    hpx::parallel::util::projection_identity>
bool none_of(Iter first, Sent last, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for no elements in the range [first, last).

Note: Complexity: At most *last - first* applications of the predicate *f*

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **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 *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

- **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.
- **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.

Returns 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 =
    hpx::parallel::util::projection_identity>
hpx::parallel::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.

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.

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.

Returns 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 Iter, typename Sent, typename F, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type any_of(ExPolicy &&policy, Iter first, Sent last, 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.

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.

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.
- **Iter** – The type of the source iterators used for the range (deduced).
- **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 *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 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.
- **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.

Returns 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 Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
bool any_of(Rng &&rng, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

Note: Complexity: At most `std::distance(begin(rng), end(rng))` applications of the predicate *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 *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

- **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.

Returns 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 Iter, typename Sent, typename F, typename Proj =  
hpx::parallel::util::projection_identity>  
bool any_of(Iter first, Sent last, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

Note: Complexity: At most `std::distance(begin(rng), end(rng))` applications of the predicate *f*

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **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 `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

- **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.
- **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.

Returns 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 =  
hpx::parallel::util::projection_identity>  
hpx::parallel::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.

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.

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.

Returns The *all_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

```
template<typename ExPolicy, typename Iter, typename Sent, typename F, typename Proj =
    hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type all_of(ExPolicy &&policy, Iter first,
    Sent last, 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.

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.

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.
- **Iter** – The type of the source iterators used for the range (deduced).
- **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 *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 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.
- **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.

Returns The *all_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

```
template<typename Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
bool all_of(Rng &&rng, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for all elements in the range *rng*.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *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 *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

- **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.

Returns The `all_of` algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

```
template<typename Iter, typename Sent, typename F, typename Proj =  
hpx::parallel::util::projection_identity>  
bool all_of(Iter first, Sent last, F &&f, Proj &&proj = Proj())
```

Checks if unary predicate *f* returns true for all elements in the range *rng*.

Note: Complexity: At most `std::distance(begin(rng), end(rng))` applications of the predicate *f*

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **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 `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

- **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.
- **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.

Returns 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.

hpx/parallel/container_algorithms/copy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter>
parallel::util::detail::algorithm_result<ExPolicy, ranges::copy_result<FwdIter1, FwdIter>>::type copy(ExPolicy
    &&policy,
    FwdIter1
    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.

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.

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 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.

Returns 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>
parallel::util::detail::algorithm_result<ExPolicy, ranges::copy_result<typename hpx::traits::range_traits<Rng>::iterator_type,
```

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.

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.

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.

Returns 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 FwdIter1, typename Sent1, typename FwdIter>
ranges::copy_result<FwdIter1, FwdIter> copy(FwdIter1 iter, Sent1 sent, FwdIter dest)
```

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **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.

- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **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.

Returns 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 FwdIter>
ranges::copy_result<typename hpx::traits::range_traits<Rng>::iterator_type, FwdIter> copy(Rng
&&rng,
FwdIter
dest)
```

Copies the elements in the range *rng* to another range beginning at *dest*.

Note: Complexity: Performs exactly std::distance(begin(rng), end(rng)) 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.
- **FwdIter** – 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.
- **dest** – Refers to the beginning of the destination range.

Returns 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, ranges::copy_n_result<FwdIter1, FwdIter2>>::type copy_n(ExPolicy
&&policy,
FwdIter
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.

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.

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.

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.

Returns 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 FwdIter1, typename Size, typename FwdIter2>
ranges::copy_n_result<FwdIter1, FwdIter2> copy_n(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.

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 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

- **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.

Returns 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 Pred, typename Proj = hpx::parallel::util::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, ranges::copy_if_result<FwdIter1, FwdIter>>::type copy_if(ExPolicy  
    &&policy,  
    FwdIter1  
    iter,  
    Sent1  
    sent,  
    FwdIter  
    dest,  
    Pred  
    &&pred  
    Proj  
    &&proj  
    =  
    Proj())
```

Copies the elements in the range, defined by [first, last) to another range beginning at *dest*. 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.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **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

- **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.
- **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.

Returns 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 FwdIter, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, ranges::copy_if_result<typename hpx::traits::range_traits<Rng>::iterator_t<Rng>, FwdIter>> copy_if(
```

Copies the elements in the range, defined by *rng* to another range beginning at *dest*. 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.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **FwdIter** – 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.
- **Proj** – The type 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** – 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.

Returns 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 FwdIter1, typename Sent1, typename FwdIter, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
ranges::copy_if_result<FwdIter1, FwdIter> copy_if(FwdIter1 iter, Sent1 sent, FwdIter dest, Pred &&pred, Proj &&proj = Proj())
```

Copies the elements in the range, defined by [first, last) to another range beginning at *dest*. The order of the elements that are not removed is preserved.

Template Parameters

- **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.
- **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

- **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.
- **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.

Returns 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 Rng, typename FwdIter, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
ranges::copy_if_result<typename hpx::traits::range_traits<Rng>::iterator_type, FwdIter> copy_if(Rng &&rng, FwdIter dest, Pred &&pred, Proj &&proj = Proj())
```

Copies the elements in the range, defined by *rng* to another range beginning at *dest*. The order of the elements that are not removed is preserved.

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.
- **FwdIter** – 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.
- **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** – 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.

Returns 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.

hpx/parallel/container_algorithms/count.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Rng, typename Proj = hpx::parallel::util::projection_identity,
typename T = typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>,
Proj>::value_type>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>
```

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to search for (deduced).
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 Iter, typename Sent, typename Proj =  
hpx::parallel::util::projection_identity, typename T = typename hpx::parallel::traits::projected<Iter,  
Proj>::value_type>  
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<Iter>::difference_type>::type count(Ex  
&  
icy  
Iter  
fir  
Se  
las  
T  
co  
&  
Pr  
&  
=  
Pr
```

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to 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 iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **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.

Returns 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 Rng, typename Proj = hpx::parallel::util::projection_identity, typename T =
typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>, Proj>::value_type>
std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_type>::difference_type count(Rng
&&rng,
T
const
&value,
Proj
&&proj
=
Proj()
```

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

Note: Complexity: Performs exactly *last - 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.
- **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

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

```
template<typename Iter, typename Sent, typename Proj = hpx::parallel::util::projection_identity,
typename T = typename hpx::parallel::traits::projected<Iter, Proj>::value_type>
std::iterator_traits<Iter>::difference_type count(Iter first, Sent 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*.

Note: Complexity: Performs exactly *last - first* comparisons.

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **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

- **first** – Refers to the beginning of the sequence 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.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

```
template<typename ExPolicy, typename Rng, typename F, typename Proj =
    hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>
```

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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<typename ExPolicy, typename Iter, typename Sent, typename F, typename Proj =
hpx::parallel::util::projection_identity>
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<Iter>::difference_type>::type count_if`

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to 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 iterators used for the range (deduced).
- **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 *count_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

`bool pred(const Type &a);`

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be

dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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<typename Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_type>::difference_type count_if(Rng
    &&rng,
    F
    &&f,
    Proj
    &&proj
    =
    Proj())
```

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note: Complexity: Performs exactly *last - first* applications of the predicate.

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 *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

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

```
template<typename Iter, typename Sent, typename F, typename Proj =
hpx::parallel::util::projection_identity>
std::iterator_traits<Iter>::difference_type count_if(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.

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **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 *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

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

hpx/parallel/container_algorithms/destroy.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Rng>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type destroy(ExPolicy &ex, Rng &r, Compare &c, Proj proj = Proj{})
```

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.

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.

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.

Returns 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 Iter, typename Sent>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type destroy(ExPolicy &&policy, Iter first,
Sent last)
```

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

The operations in the parallel *destroy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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.
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns 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 Rng>
hpx::traits::range_iterator<Rng>::type destroy(Rng &&rng)
```

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

Note: Complexity: Performs exactly *last - first* operations.

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 sequence of elements the algorithm will be applied to.

Returns The *destroy* algorithm returns `void`.

```
template<typename Iter, typename Sent>
inline Iter destroy(Iter first, Sent last)
```

Move objects to uninitialized memory.

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

Note: Complexity: Performs exactly *last - first* operations.

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **Iter** – : iterator to the elements
- **Value** – : typename of the object to construct

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.
- **ptr** – [in] : pointer to the memory where to construct the object
- **R** – [in] : range to move

Returns The *destroy* algorithm returns `void`.

```
template<typename ExPolicy, typename FwdIter, typename Size>
hpx::parallel::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.

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.

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.

Returns The *destroy_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

```
template<typename FwdIter, typename Size>
FwdIter destroy_n(FwdIter first, Size count)
```

Destroys objects of type *typename iterator_traits<ForwardIt>::value_type* in the range [first, first + count).

Note: Complexity: Performs exactly *count* operations, 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 this algorithm 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.

Returns The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

[hpx/parallel/container_algorithms/ends_with.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred =  
ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 =  
parallel::util::projection_identity>  
bool ends_with(Iter1 first1, Sent1 last1, Iter2 first2, Sent2 last2, Pred &&pred = Pred(), Proj1 &&proj1  
= Proj1(), Proj2 &&proj2 = Proj2())
```

Checks whether the second range defined by [first1, last1) matches the suffix of the first range defined by [first2, last2)

The assignments in the parallel *ends_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **Iter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate is invoked.

Returns 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 Pred = ranges::equal_to, typename Proj1 = parallel::util::projection_identity,  
typename Proj2 = parallel::util::projection_identity>
```

```
parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy &&policy, FwdIter1
    first1, Sent1 last1, FwdIter2
    first2, Sent2 last2, Pred
    &&pred = Pred(), Proj1
    &&proj1 = Proj1(), Proj2
    &&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.

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.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **FwdIter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the 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.

Returns 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 = ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
bool ends_with(Rng1 &&rng1, Rng2 &&rng2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(),
                Proj2 &&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*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate is invoked.

Returns 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 = ranges::equal_to,
        typename Proj1 = parallel::util::projection_identity, typename Proj2 =
        parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy &&policy, Rng1
    &&rng1, Rng2 &&rng2,
    Pred &&pred = Pred(),
    Proj1 &&proj1 = Proj1(),
    Proj2 &&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.

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.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns 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.

hpx/parallel/container_algorithms/equal.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,  
typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2  
= hpx::parallel::util::projection_identity>  
hpx::parallel::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.

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: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *f*.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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 = equal_to, typename  
Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =  
hpx::parallel::util::projection_identity>  
hpx::parallel::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.

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: Complexity: At most $last1 - first1$ applications of the predicate *f*.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

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

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
bool equal(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.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *f*.

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.

Template Parameters

- **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*.

structible. 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

- **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.

Returns 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 Rng1, typename Rng2, typename Pred = equal_to, typename Proj1 =
    hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
bool equal(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.

Note: Complexity: At most *last1 - first1* applications of the predicate *f*.

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.

Template Parameters

- **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

- **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.

Returns The `equal` algorithm returns true if the elements in the two ranges are equal, otherwise it returns false.

hpx/parallel/container_algorithms/exclusive_scan.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename InIter, typename Sent, typename OutIter, typename T = typename
std::iterator_traits<InIter>::value_type, typename Op = std::plus<T>>
exclusive_scan_result<InIter, OutIter> exclusive_scan(InIter first, Sent last, OutIter dest, T init, Op
&&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 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.

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

Note: GENERALIZED_NONCOMMUTATIVE_SUM(*op*, *a*1, ..., *a*N) 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 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 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

- **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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename T
= typename std::iterator_traits<FwdIter1>::value_type, typename Op = std::plus<T>>
parallel::util::detail::algorithm_result<ExPolicy, exclusive_scan_result<FwdIter1, FwdIter2>>::type exclusive_scan(ExPolicy
&Op, 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, ..., *(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.

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.

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

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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.

```
template<typename Rng, typename O, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type, typename Op = std::plus<T>>
exclusive_scan_result<traits::range_iterator_t<Rng>, O> exclusive_scan(Rng &&rng, O dest, T init,
Op &&cop = Op())
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the i th input element in the i th sum.

Note: Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

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 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.
- **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

- **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.

Returns 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.

```
template<typename ExPolicy, typename Rng, typename O, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type, typename Op = std::plus<T>>
```

`parallel::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.

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.

Note: Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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.

hpx/parallel/container_algorithms/fill.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Rng, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type fill(
```

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.

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.

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.

Returns 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 Iter, typename Sent, typename T = typename std::iterator_traits<Iter>::value_type>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type fill(ExPolicy &&policy, Iter first,
Sent last, 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.

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.

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 iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **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 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.
- **value** – The value to be assigned.

Returns 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 Rng, typename T = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
hpx::traits::range_iterator_t<Rng> fill(Rng &&rng, T const &value)
```

Assigns the given value to the elements in the range [first, last).

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.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns *void*.

```
template<typename Iter, typename Sent, typename T = typename
std::iterator_traits<Iter>::value_type>
Iter fill(Iter first, Sent last, T const &value)
```

Assigns the given value to the elements in the range [first, last).

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **T** – The type of the value to be assigned (deduced).

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.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns *void*.

```
template<typename ExPolicy, typename Rng, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, hpx::traits::range_iterator_t<Rng>> fill_n(ExPolicy
&&policy,
Rng
&&rng,
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.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*).

```
template<typename ExPolicy, typename FwdIter, typename Size, typename T = typename std::iterator_traits<FwdIter>::value_type>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type fill_n(ExPolicy &&policy,
                                                               FwdIter 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.

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.

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 for the range (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.
- **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.

Returns The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*).

```
template<typename Rng, typename T = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
hpx::traits::range_traits<Rng>::iterator_type fill_n(Rng &&rng, 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.

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 sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill_n* algorithm returns an output iterator that compares equal to last.

```
template<typename FwdIter, typename Size, typename T = typename  
std::iterator_traits<FwdIter>::value_type>  
FwdIter fill_n(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.

Note: Complexity: Performs exactly *count* assignments, for count > 0.

Template Parameters

- **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

- **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.

Returns The *fill_n* algorithm returns an output iterator that compares equal to last.

hpx/parallel/container_algorithms/find.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter, typename Sent, typename Proj =  
hpx::parallel::util::projection_identity, typename T = typename hpx::parallel::traits::projected<Iter,  
Proj>::value_type>  
hpx::parallel::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.

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.

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.

Returns 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 Proj = hpx::parallel::util::projection_identity,
typename T = typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>,
Proj>::value_type>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::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.

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.

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.

Returns 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 Iter, typename Sent, typename Proj = hpx::parallel::util::projection_identity,
        typename T = typename hpx::parallel::traits::projected<Iter, Proj>::value_type>
Iter find(Iter first, Sent last, T const &val, Proj &&proj = Proj())
```

Returns the first element in the range [first, last) that is equal to value

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **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

- **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.

Returns 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 Rng, typename Proj = hpx::parallel::util::projection_identity, typename T =
typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>, Proj>::value_type>
hpx::traits::range_iterator<Rng>::type find(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.

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.

Note: Complexity: At most last - first applications of the operator==().

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 find (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.
- **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.

Returns 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 Iter, typename Sent, typename Pred, typename Proj =
hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type find_if(ExPolicy &&policy, Iter
first, Sent last, Pred &&pred,
Proj &&proj = Proj())
```

Returns the first element in the range [first, last) for which predicate *pred* 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.

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.

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.

- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an 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*.
- **Proj** – The type 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.
- **pred** – 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*.

- **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.

Returns 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 Rng, typename Pred, typename Proj =
hpx::parallel::util::projection_identity>
hpx::parallel::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type find_if(Ex
&c
icy
Rn
&c
Pr
&c
Pr
&c
=
```

Returns the first element in the range *rng* for which predicate *pred* 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.

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.

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.
- **Rng** – 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 the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* 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.
- **pred** – 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*.

- **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.

Returns 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 Iter, typename Sent, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
Iter find_if(Iter first, Sent last, Pred &&pred, Proj &&proj = Proj())
```

Returns the first element in the range [first, last) for which predicate *pred* returns true

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an 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*.
- **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 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.
- **pred** – 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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 Rng, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng>::type find_if(Rng &&rng, Pred &&pred, Proj &&proj = Proj())
```

Returns the first element in the range *rng* for which predicate *pred* returns true

Note: Complexity: At most last - first 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.
- **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*.
- **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** – 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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 Iter, typename Sent, typename Pred, typename Proj =
hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type find_if_not(ExPolicy &&policy,
Iter first, Sent last, Pred &&pred, Proj &&proj = Proj())
```

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.

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.

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.
- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an 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*.
- **Proj** – The type 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.
- **pred** – 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*.

- **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.

Returns 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 Rng, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
```

hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type **find_if_no**

Returns the first element in the range *rng* 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.

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.

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.
- **Rng** – 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 the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* 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.
- **pred** – 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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 Iter, typename Sent, typename Pred, typename Proj =  
    hpx::parallel::util::projection_identity>  
Iter find_if_not(Iter first, Sent last, Pred &&pred, Proj &&proj = Proj())
```

Returns the first element in the range [first, last) for which predicate *f* returns false

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an 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*.
- **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 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.
- **pred** – 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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 Rng, typename Pred, typename Proj = hpx::parallel::util::projection_identity>  
    hpx::traits::range_iterator<Rng>::type find_if_not(Rng &&rng, Pred &&pred, Proj &&proj = Proj())
```

Returns the first element in the range *rng* for which predicate *f* returns false

Note: Complexity: At most last - first 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.
- **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*.
- **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** – 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`.

- **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.

Returns 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 Rng1, typename Rng2, typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_end(
```

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.

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*.

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.
- **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<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.

Returns The *find_end* algorithm returns a `hpx::future<iterator_t<Rng>>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *iterator_t<Rng>* otherwise. The *find_end* algorithm returns an iterator to the beginning of the last subsequence *rng2* in range *rng*. If the length of the subsequence *rng2* is greater than the length of the range *rng*, *end(rng)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng)* is also returned.

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,
        typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =
        hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter1>::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.

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*.

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.

Returns 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 Rng1, typename Rng2, typename Pred = equal_to, typename Proj1 =
    hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng1>::type find_end(Rng1 &&rng1, 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.

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

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

- **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

- **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<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.

Returns 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 Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
Iter1 find_end(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.

This overload of `find_end` is available if the user decides to provide the algorithm their own predicate `op`.

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

Template Parameters

- **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

- **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:

<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 ob-

jects 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.

Returns 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 = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_first
```

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.

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*.

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.

Returns 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 Iter1, typename Sent1, typename Iter2, typename Sent2,
        typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2
        = hpx::parallel::util::projection_identityhpx::parallel::util::detail::algorithm_result<ExPolicy, Iter1>::type find_first_of(ExPolicy &&policy,
Iter1 first1, Sent1
last1, Iter2 first2,
Sent2 last2, Pred
&&op = Pred(),
Proj1 &&proj1 = Proj1(), Proj2
&&proj2 = Proj2())
```

Searches the range [first1, last1) for any elements in the range [first2, last2). Uses binary predicate *p* to compare elements

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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*.

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

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj1** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements in *rng1*.
- **Proj2** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements in *rng2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
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.

Returns 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 Rng1, typename Rng2, typename Pred = equal_to, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng1>::type find_first_of(Rng1 &&rng1, Rng2 &&rng2, Pred &&op =
Pred(), Proj1 &&proj1 = Proj1(), Proj2
&&proj2 = Proj2())
```

Searches the range `rng1` for any elements in the range `rng2`. Uses binary predicate `p` to compare elements

This overload of `find_first_of` is available if the user decides to provide the algorithm their own predicate `op`.

Note: Complexity: at most $(S*N)$ comparisons where $S = \text{distance}(\text{begin}(rng2), \text{end}(rng2))$ and $N = \text{distance}(\text{begin}(rng1), \text{end}(rng1))$.

Template Parameters

- **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

- **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.

Returns 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 Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred =  
equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =  
hpx::parallel::util::projection_identity>  
Iter1 find_first_of(Iter1 first1, Sent1 last1, Iter2 first2, Sent2 last2, Pred &&op = Pred(), Proj1  
&&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Searches the range [first1, last1) for any elements in the range [first2, last2). Uses binary predicate *p* to compare elements

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (*S***N*) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **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

- **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:

<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 *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.

Returns 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.

hpx/parallel/container_algorithms/for_each.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename InIter, typename Sent, typename F, typename Proj =
    hpx::parallel::util::projection_identity>
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.

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

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

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 *hpx::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.

Returns {last, HPX_MOVE(f)} where last is the iterator corresponding to the input sentinel last.

```
template<typename Rng, typename F, typename Proj = hpx::parallel::util::projection_identity>
for_each_result<typename hpx::traits::range_iterator<Rng>::type, F> for_each(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.

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

Note: Complexity: Applies *f* exactly *size(rng)* times.

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 *hpx::identity*

Parameters

- **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.

Returns {std::end(rng), HPX_MOVE(f)}

```
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 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.

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.

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.

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

Template Parameters

- **ExPolicy** – The type of the execution 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 *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have **const&**. The type *Type* must be such that an object of type *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.

Returns 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 Rng, typename F, typename Proj =
hpx::parallel::util::projection_identity>
```

hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type **for_each**(*E*, *f*, *R*, *&P*, *P*)

Applies *f* to the result of dereferencing every iterator in the given range *rng*.

If *f* returns a result, the result is ignored.

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.

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.

Note: Complexity: Applies *f* exactly *size(rng)* times.

Template Parameters

- **ExPolicy** – The type of the execution 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 *hpx::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.

Returns 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 InIter, typename Size, typename F, typename Proj =
hpx::parallel::util::projection_identity>
for_each_n_result<InIter, F> for_each_n(InIter 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.

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.

Note: Complexity: Applies *f* exactly *count* times.

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 *hpx::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.

Returns It returns *last*.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename F, typename Proj =
hpx::parallel::util::projection_identity>
hpx::parallel::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.

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.

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.

Note: Complexity: Applies f exactly $count$ times.

Template Parameters

- **ExPolicy** – The type of the execution 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 *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have **const&**. The type *Type* must be such that an object of type *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.

Returns 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*.

hpx/parallel/container_algorithms/for_loop.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

namespace **experimental**

Functions

```
template<typename ExPolicy, typename Iter, typename Sent, typename ...Args>
hpx::parallel::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.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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 ...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`.

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:

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.

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.

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.

```
template<typename ExPolicy, typename R, typename ...Args>
hpx::parallel::util::detail::algorithm_result<ExPolicy>::type for_loop(ExPolicy &&policy, R
&&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`.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.
- **R** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Args** – A parameter pack, its last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of 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.

Returns 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`.

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:

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.

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.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **rng** – Refers to the 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.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **S**, typename ...**Args**>

```
parallel::util::detail::algorithm_result<ExPolicy>::type for_loop_strided(ExPolicy &&policy,  
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.

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`.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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 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*.

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:

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.

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.

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, 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 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.

```
template<typename ExPolicy, typename Rng, typename S, typename ...Args>
hpx::parallel::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 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`.

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:

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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*.

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:

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.

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.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **rng** – Refers to the 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.

hpx/parallel/container_algorithms/generate.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Rng, typename F>
hpx::parallel::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.

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.

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*.

Returns 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 Fhpx::parallel::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.

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.

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*.

Returns 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 Rng, typename F>
hpx::traits::range_iterator<Rng>::type generate(Rng &&rng, F &&f)
```

Assign each element in range [first, last) a value generated by the given function object f

Note: Complexity: Exactly *distance(first, last)* invocations of f and 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 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

- **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*.

Returns The *replace_if* algorithm returns *last*.

```
template<typename Iter, typename Sent, typename F>
Iter generate(Iter first, Sent last, F &&f)
```

Assign each element in range [first, last) a value generated by the given function object f

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters

- **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

- **first** – Refers to the beginning of the sequence 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*.

Returns The *replace_if* algorithm returns *last*.

```
template<typename ExPolicy, typename FwdIter, typename Size, typename F>
hpx::parallel::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.

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.

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.
- **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 *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*.

Returns 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 FwdIter, typename Size, typename F>
FwdIter generate_n(FwdIter first, Size count, F &&f)
```

Assigns each element in range [first, first+count) a value generated by the given function object *g*.

Note: Complexity: Exactly *count* invocations of *f* and assignments, for *count* > 0.

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.
- **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

- **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*.

Returns The *replace_if* algorithm returns *last*.

hpx/parallel/container_algorithms/includes.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,
typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identityhpx::parallel::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.

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.

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.
- **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.

Returns 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 Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred = hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
bool includes(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`.

Note: At most $2*(N1+N2-1)$ comparisons, where $N1 = \text{std::distance(first1, last1)}$ and $N2 = \text{std::distance(first2, last2)}$.

Template Parameters

- **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

- **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.

Returns 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 =
    hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename
    Proj2 = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type includes(ExPolicy &&policy, Rng1
    &&rng1, Rng2 &&rng2,
    Pred &&op = Pred(),
    Proj1 &&proj1 = Proj1(),
    Proj2 &&proj2 = Proj2())
```

Returns true if every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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.

Returns 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 Rng1, typename Rng2, typename Pred = hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
bool includes(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*.

Note: At most $2*(N1+N2-1)$ comparisons, where $N1 = \text{std::distance(first1, last1)}$ and $N2 =$

std::distance(first2, last2).

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 *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

- **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.

Returns 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.

hpx/parallel/container_algorithms/inclusive_scan.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename InIter, typename Sent, typename OutIter, typename Op = std::plus<typename std::iterator_traits<InIter>::value_type>>
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.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the i th input element in the i th sum.

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

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 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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename Op = std::plus<typename std::iterator_traits<FwdIter1>::value_type>>
```

```
parallel::util::detail::algorithm_result<ExPolicy, inclusive_scan_result<FwdIter1, FwdIter2>>::type inclusive_scan(ExPolicy
&&
icy
Fw
firs
Ser
last
Fw
des
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 with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
 - GENERALIZED_NONCOMMUTATIVE_SUM($op, a1, \dots, aK$)
 - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for 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.

Returns 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.

```
template<typename Rng, typename O, typename Op = std::plus<typename hpx::traits::range_traits<Rng>::value_type>>
inclusive_scan_result<hpx::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, \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.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i th input element in the i th sum.

Note: Complexity: $O(last - first)$ applications of the predicate `op`.

Note: `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, \dots, aN)` is defined as:

- $a1$ when N is 1
- `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, \dots, aK)`
 - `GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, \dots, aN)` where $1 < K+1 = M \leq N$.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced). 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.

Returns 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.

```
template<typename ExPolicy, typename Rng, typename O, typename Op = std::plus<typename hpx::traits::range_traits<Rng>::value_type>>
```

```
parallel::util::detail::algorithm_result<ExPolicy, inclusive_scan_result<hpx::traits::range_iterator_t<Rng>, O>>::type incl
```

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.

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.

Note: Complexity: $O(last - first)$ applications of the predicate `op`.

Note: `GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN)` is defined as:

- $a1$ when N is 1
 - $GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, aK)$
 - $GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, \dots, aN)$ where $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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.

Returns 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

```
template<typename InIter, typename Sent, typename OutIter, typename Op, typename T = typename std::iterator_traits<InIter>::value_type>
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.

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.

Note: Complexity: $O(last - first)$ applications of the predicate `op`.

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 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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename Op, typename T = typename std::iterator_traits<FwdIter1>::value_type>
parallel::util::detail::algorithm_result<ExPolicy, inclusive_scan_result<FwdIter1, FwdIter2>>::type inclusive_scan(ExPolicy policy, InIter first, OutIter last, Op op, FwdIter2 dest, 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 with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

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.

Note: Complexity: $O(last - first)$ applications of the predicate `op`.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) is defined as:

- a1 when N is 1
 - op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN)) where 1 < K+1 = M <= N.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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

- **policy** – The execution policy to use for 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.

- **init** – The initial value for the generalized sum.

Returns The *inclusive_scan* algorithm returns a *hpx::future<util::in_out_result<InIter, OutIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result<InIter, OutIter>* 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.

```
template<typename Rng, typename O, typename Op, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
inclusive_scan_result<hpx::traits::range_iterator_t<Rng>, O> inclusive_scan(Rng &&rng, O dest,
Op &&op, T init)
```

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, ..., *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

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.

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

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_N) is defined as:

- a_1 when N is 1
 - $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_M, \dots, a_N))$ where $1 < K+1 = M \leq N$.
-

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced). 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.

Returns 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.

```
template<typename ExPolicy, typename Rng, typename O, typename Op, typename T = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
parallel::util::detail::algorithm_result<ExPolicy, inclusive_scan_result<hpx::traits::range_iterator_t<Rng>, O>>::type incl
```

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 with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **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.
- **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.

Returns 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

`hpx/parallel/container_algorithms/is_heap.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Rng, typename Comp = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::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.

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.

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.

Returns 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 =  
hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>  
hpx::parallel::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.

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.

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.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* is invoked.

Returns 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 Rng, typename Comp = hpx::parallel::v1::detail::less, typename Proj =  
hpx::parallel::util::projection_identity>  
bool is_heap(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

- **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

- **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.

Returns The *is_heap* algorithm returns *bool*. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

```
template<typename Iter, typename Sent, typename Comp = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
bool is_heap(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

- **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

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap* algorithm returns *bool*. 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type is_heap_un
```

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.

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.

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.

Returns 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 Iter, typename Sent, typename Comp =
  hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type is_heap_until(ExPolicy &&policy,
  Iter first, Sent last,
  Comp &&comp =
  Comp(), Proj
  &&proj = Proj())
```

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

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.

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.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* is invoked.

Returns 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 Rng, typename Comp = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng>::type is_heap_until(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

- **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

- **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.

Returns The *is_heap_until* algorithm returns *RandIter*. 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 Iter, typename Sent, typename Comp = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
Iter is_heap_until(Iter first, Sent last, Comp &&comp = Comp(), Proj &&proj = Proj())
```

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [*first*, *it*) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **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

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* is invoked.

Returns The *is_heap_until* algorithm returns *RandIter*. 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.

hpx/parallel/container_algorithms/is_partitioned.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename FwdIter, typename Sent, typename Pred, typename Proj =
  hpx::parallel::util::projection_identity

```

```
bool is_partitioned(FwdIter first, Sent last, Pred &&pred, Proj &&proj = Proj())
```

Determines if the range [*first*, *last*) is partitioned.

Note: Complexity: at most (*N*) predicate evaluations where *N* = distance(*first*, *last*).

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 source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- **Pred** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::parallel::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 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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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<typename ExPolicy, typename FwdIter, typename Sent, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, bool>::type is_partitioned(ExPolicy &&policy,
FwdIter first, Sent last,
Pred &&pred, Proj
&&proj = Proj())
```

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.

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.

Note: Complexity: at most (N) predicate evaluations where N = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **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). *Pred* must be *CopyConstructible* when using a parallel policy.
- **Proj** – The type of an optional projection function. This defaults to *hpx::parallel::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 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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_partitioned* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_partitioned* algorithm returns true if each element in the sequence for which pred returns true precedes those for which pred returns false. Otherwise *is_partitioned* returns false. If the range [first, last) contains less than two elements, the function is always true.

```
template<typename Rng, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
bool is_partitioned(Rng &&rng, Pred &&pred, Proj &&proj = Proj())
```

Determines if the range rng is partitioned.

Note: Complexity: at most (N) predicate evaluations where $N = \text{std::size(rng)}$.

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).
- **Proj** – The type of an optional projection function. This defaults to *hpx::parallel::util::projection_identity*.

Parameters

- **rng** – Refers to the sequence of elements 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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 rng contains less than two elements, the function is always true.

```
template<typename ExPolicy, typename Rng, typename Pred, typename Proj =
hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, bool>::type is_partitioned(ExPolicy &&policy, Rng
&&rng, Pred &&pred,
Proj &&proj = Proj())
```

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.

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.

Note: Complexity: at most (N) predicate evaluations where $N = \text{std::size(rng)}$.

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). *Pred* must be *CopyConstructible* when using a parallel policy.
- **Proj** – The type of an optional projection function. This defaults to `hpx::parallel::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 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*.

- **proj** – 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.

Returns 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 *rng* contains less than two elements, the function is always true.

hpx/parallel/container_algorithms/is_sorted.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **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.

Returns 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.

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.

Note: Complexity: at most (N+S-1) comparisons where N = `distance(first, last)`. S = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.

- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::less<>`
- **Proj** – The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
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.

Returns The *is_sorted* algorithm returns a `hpx::future<bool>` if the execution policy is of type *task_execution_policy* and returns `bool` otherwise. The *is_sorted* algorithm returns a `bool` if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

```
template<typename Rng, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
bool is_sorted(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.

Returns 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::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type is_sorted(ExPolicy &&policy, Rng &&rng, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Determines if the range `rng` is sorted. Uses `pred` to compare elements.

The comparison operations in the parallel `is_sorted` algorithm invoked with an execution policy object of type `sequenced_policy` executes in sequential order in the calling thread.

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.

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.

Returns 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::util::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.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- **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.

Returns 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::util::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type is_sorted_until(ExPolicy
    &&policy,
    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.

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.

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.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for *FwdIter*.
- **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

<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*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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_identity>
hpx::traits::range_iterator<Rng>::type is_sorted_until(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.

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. 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

- **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.

Returns The *is_sorted_until* returns *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::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type is_sorted_
```

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.

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.

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` is invoked.

Returns 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.

hpx/parallel/container_algorithms/lexicographical_compare.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename InIter1, typename Sent1, typename InIter2, typename Sent2, typename Proj1
= hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity,
typename Pred = hpx::parallel::v1::detail::less>
bool lexicographical_compare(InIter1 first1, Sent1 last1, InIter2 first2, Sent2 last2, Pred &&pred =
    Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked 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 $N1 = \text{std}::\text{distance}(\text{first1}, \text{last1})$ and $N2 = \text{std}::\text{distance}(\text{first2}, \text{last2})$.

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*
-

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.

Returns 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 FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity, typename Pred = hpx::parallel::v1::detail::less>
parallel::util::detail::algorithm_result<ExPolicy, bool>::type lexicalographical_compare(ExPolicy &&policy, FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel `lexicalographical_compare` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

The comparison operations in the parallel `lexicalographical_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: Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 =$

std::distance(first1, last) and N2 = std::distance(first2, last2).

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*
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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, typename Pred =
hpx::parallel::v1::detail::less>
bool lexicographical_compare(Rng1 &&rng1, Rng2 &&rng2, Pred &&pred = Pred(), Proj1
&&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Checks if the first range rng1 is lexicographically less than the second range rng2. uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: At most $2 * \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std::distance}(\text{std::begin(rng1)}, \text{std::end(rng1)})$ and $N2 = \text{std::distance}(\text{std::begin(rng2)}, \text{std::end(rng2)})$.

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*
-

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.

Returns 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_identity, typename Proj2 = hpx::parallel::util::projection_identity,  
    typename Pred = hpx::parallel::v1::detail::less>  
parallel::util::detail::algorithm_result<ExPolicy, bool>::type lexicographical_compare(ExPolicy  
    &&policy,  
    Rng1  
    &&rng1,  
    Rng2  
    &&rng2,  
    Pred  
    &&pred =  
    Pred(),  
    Proj1  
    &&proj1 =  
    Proj1(),  
    Proj2  
    &&proj2 =  
    Proj2())
```

Checks if the first range *rng1* is lexicographically less than the second range *rng2*. uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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: 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 $N2 = \text{std}::distance(\text{std}::begin(rng2), \text{std}::end(rng2))$.

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*
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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.

hpx/parallel/container_algorithms/make_heap.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter, typename Sent, typename Comp, typename Proj =
hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type make_heap(ExPolicy &&policy, Iter
first, Sent last, 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.

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.

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.
- **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.
- **first** – Refers to the beginning of the sequence 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** – 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.

Returns 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 Comp, typename Proj =  
hpx::parallel::util::projection_identity>  
hpx::parallel::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type make_heap(
```

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.

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.

Note: Complexity: at most $(3*N)$ comparisons where $N = \text{distance(first, last)}$.

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.
- **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** – 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.

Returns 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 Iter, typename Sent, typename Proj =
hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type make_heap(ExPolicy &&policy, Iter
first, Sent last, 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.

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.

Note: Complexity: at most $(3*N)$ comparisons where $N = \text{distance(first, last)}$.

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.
- **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.
- **Proj** – The type 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.
- **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.

Returns 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 = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type make_heap(
```

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.

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.

Note: Complexity: at most $(3*N)$ comparisons where $N = \text{distance(first, last)}$.

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.
- **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

- **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.
- **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.

Returns 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 Iter, typename Sent, typename Comp, typename Proj =
  hpx::parallel::util::projection_identity>
Iter make_heap(Iter first, Sent last, Comp &&comp, Proj &&proj = Proj{ })
```

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most $(3*N)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- **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

- **first** – Refers to the beginning of the sequence 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** – 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.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

```
template<typename Rng, typename Comp, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng>::type make_heap(Rng &&rng, Comp &&comp, Proj &&proj =
  Proj{ })
```

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most $(3*N)$ comparisons where $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an 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.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

```
template<typename Iter, typename Sent, typename Proj = hpx::parallel::util::projection_identity>
Iter make_heap(Iter first, Sent last, Proj &&proj = Proj{})
```

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- **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*.
- **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.
- **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.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

```
template<typename Rng, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator<Rng>::type make_heap(Rng &&rng, Proj &&proj = Proj{})
```

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where $N = \text{distance}(\text{first}, \text{last})$.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an 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.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

hpx/parallel/container_algorithms/merge.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename Iter3, typename Comp =
    hpx::ranges::less, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =
    hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::merge_result<typename hpx::traits::range_iterator<Rng1>, typename hpx::traits::range_iterator<Rng2>, Iter3, Comp, Proj1, 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.

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.

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.
- **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.
- **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 *Iter1* and *Iter2* 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.

Returns 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 Iter1, typename Sent1, typename Iter2, typename Sent2,
         typename Iter3, typename Comp = hpx::ranges::less, typename Proj1 =
         hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

```

hpx::parallel::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
    &&proj =
    Proj1(),
    Proj2
    &&proj =
    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.

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.

Note: Complexity: Performs $O(\text{std}::\text{distance}(first1, last1) + \text{std}::\text{distance}(first2, last2))$ 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.
- **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 *Iter1* and *Iter2* 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.

Returns 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 Rng1, typename Rng2, typename Iter3, typename Comp = hpx::ranges::less,
         typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =
         hpx::parallel::util::projection_identity>
```

`hpx::ranges::merge_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type, Proj1, 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.

Note: Complexity: Performs $O(\text{std}::\text{distance}(first1, last1) + \text{std}::\text{distance}(first2, last2))$ applications of the comparison *comp* and the each projection.

Template Parameters

- **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.
- **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

- **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 *Iter1* and *Iter2* 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.

Returns The *merge* algorithm returns *merge_result<Iter1, Iter2, Iter3>*. 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 Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3,
typename Comp = hpx::ranges::less, typename Proj1 = hpx::parallel::util::projection_identity, typename
Proj2 = hpx::parallel::util::projection_identity>
hpx::ranges::merge_result<Iter1, Iter2, Iter3> merge(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.

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

- **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 *Copy-Constructible*. 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

- **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 `Iter1` and `Iter2` 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.

Returns The `merge` algorithm returns `merge_result<Iter1, Iter2, Iter3>`. The `merge` algorithm returns the tuple of the source iterator `last1`, the source iterator `last2`, the destination iterator to the end of the `dest` range.

```
template<typename ExPolicy, typename Rng, typename Iter, typename Comp = hpx::ranges::less,
typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type inplace_merge(ExPolicy &&policy,
Rng &&rng, Iter
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.

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.

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.
- **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.
- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `inplace_merge` requires `Comp` to meet the requirements of `CopyConstructible`. This defaults to `std::less<>`
- **Proj** – The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the range of elements the algorithm will be applied to.
- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

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

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter* 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.

Returns 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 Iter, typename Sent, typename Comp = hpx::ranges::less,
          typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type inplace_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.

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.

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 `Iter` 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.

Returns 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 Rng, typename Iter, typename Comp = hpx::ranges::less, typename Proj = hpx::parallel::util::projection_identity>
Iter inplace_merge(Rng &&rng, Iter 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.

Note: Complexity: Performs $O(\text{std}::\text{distance}(\text{first}, \text{last}))$ applications of the comparison `comp` and the each 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 random access iterator.
- **Iter** – 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

- **rng** – Refers to the range of elements the algorithm will be applied to.
- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **comp** – `comp` is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

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

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `Iter` 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.

Returns The *inplace_merge* algorithm returns *Iter*. The *inplace_merge* algorithm returns the source iterator *last*

```
template<typename Iter, typename Sent, typename Comp = hpx::ranges::less, typename Proj = hpx::parallel::util::projection_identity>
```

Iter **inplace_merge**(*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.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **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

- **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 *Iter* 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.

Returns The *inplace_merge* algorithm *Iter*. The *inplace_merge* algorithm returns the source iterator *last*

hpx/parallel/container_algorithms/minmax.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename FwdIter, typename Sent, typename F = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identityFwdIter 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 $N = \text{std::distance}(\text{first}, \text{last})$.

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 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.

Returns 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 = hpx::parallel::v1::detail::less, 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 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.

Returns 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 =  
hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>  
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter> min_element(ExPolicy &&policy, 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 invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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 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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, hpx::traits::range_iterator_t<Rng>> min_element(ExPolicy
    &&policy,
    Rng
    &&rng,
    F
    &&f
    =
    F(),
    Proj
    &&proj
    =
    Proj())
```

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

Note: Complexity: Exactly *max(N-1, 0)* comparisons, where *N* = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *min_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

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

The signature does not need to have **const &**, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity

```

```
FwdIter max_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 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* = *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 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.

Returns 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 = hpx::parallel::v1::detail::less, 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* = 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 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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity> hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter> max_element(ExPolicy &&policy, 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 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.

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.

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 *max_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to *std::less*. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

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

The signature does not need to have *const &*, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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** = *hpx::parallel::v1::detail::less*, typename **Proj** = *hpx::parallel::util::projection_identity*>

```
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, hpx::traits::range_iterator_t<Rng>> max_element(ExPolicy
&&policy,
Rng
&&rng,
F
&&f
=
F(),
Proj
&&proj
=
Proj())
```

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

Note: Complexity: Exactly *max(N-1, 0)* comparisons, where *N* = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *max_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

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

The signature does not need to have **const &**, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identityFwdIter> 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 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 *f* is invoked.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
minmax_element_result<hpx::traits::range_iterator_t<Rng>> minmax_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 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 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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, minmax_element_result<FwdIter>> minmax_element(ExPolicy
    &&pol-
    icy,
    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 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.

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.

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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
hpx::parallel::util::algorithm_result_t<ExPolicy, minmax_element_result<hpx::traits::range_iterator_t<Rng>>> minmax_
```

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.

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.

Note: Complexity: At most $\max(\text{floor}(3/2*(N-1)), 0)$ applications of the predicate, where $N = \text{std::distance(first, last)}$.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **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.

Returns 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.

hpx/parallel/container_algorithms/mismatch.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,  
typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2  
= hpx::parallel::util::projection_identity

```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, mismatch_result<Iter1, Iter2>>::type mismatch(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 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.

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: 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.

Note: The two ranges are considered mismatch if, for every iterator i in the range $[\text{first1}, \text{last1})$, $*i \neq *(\text{first2} + (i - \text{first1}))$. This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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 = equal_to, typename  

Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =  

hpx::parallel::util::projection_identity>
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, mismatch_result<typename hpx::traits::range_traits<Rng1>::iterator_`

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.

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: 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.

Returns The *mismatch* algorithm returns a `hpx::future<std::pair<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `std::pair<FwdIter1, FwdIter2>` otherwise. The *mismatch* algorithm returns the first mismatching pair of elements from two ranges: one defined by `[first1, last1)` and another defined by `[first2, last2)`.

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred = equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
mismatch_result<Iter1, Iter2> mismatch(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 mismatch to the range `[first2, last2)`, and false otherwise.

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.

Note: The two ranges are considered mismatch if, for every iterator *i* in the range `[first1, last1)`, $*i$ mismatch $*(\text{first2} + (\text{i} - \text{first1}))$. This overload of *mismatch* uses operator`==` to determine if two elements are mismatch.

Template Parameters

- **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

- **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.

Returns The *mismatch* algorithm returns `bool`. 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 Rng1, typename Rng2, typename Pred = equal_to, typename Proj1 =
    hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
mismatch_result<typename hpx::traits::range_traits<Rng1>::iterator_type, typename hpx::traits::range_traits<Rng2>::iterator_type>
```

Returns `std::pair` with iterators to the first two non-equivalent elements.

Note: Complexity: At most `last1 - first1` applications of the predicate *f*.

Template Parameters

- **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

- **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.

Returns The *mismatch* algorithm returns `std::pair<FwdIter1, FwdIter2>`. The *mismatch* algorithm returns the first mismatching pair of elements from two ranges: one defined by `[first1, last1)` and another defined by `[first2, last2)`.

hpx/parallel/container_algorithms/move.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2hpx::parallel::util::detail::algorithm_result<ExPolicy, move_result<Iter1, Iter2>>::type move(ExPolicy
    &&policy,
    Iter1 first,
    Sent1 last,
    Iter2 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.

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.

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.
- **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.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence 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.

Returns 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 Iter2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, move_result<typename hpx::traits::range_iterator<Rng>::type, Iter2>>
```

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.

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.

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.

- **Iter2** – The type of the source iterators used for the second 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.

Returns 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 Iter1, typename Sent1, typename Iter2>
move_result<Iter1, Iter2> move(Iter1 first, Sent1 last, Iter2 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.

Note: Complexity: Performs exactly `std::distance(begin(rng), end(rng))` assignments.

Template Parameters

- **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.

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.

Returns The *move* algorithm returns `ranges::move_result<iterator_t<Rng>, FwdIter2>`. 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 Rng, typename Iter2>
move_result<typename hpx::traits::range_iterator<Rng>::type, Iter2> move(Rng &&rng, Iter2 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.

Note: Complexity: Performs exactly `std::distance(begin(rng), end(rng))` 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.
- **Iter2** – The type of the source iterators used for the second 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.
- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns a `ranges::move_result<iterator_t<Rng>, FwdIter2>`.

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.

hpx/parallel/container_algorithms/nth_element.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename RandomIt, typename Sent, typename Pred = hpx::parallel::v1::detail::less,
          typename Proj = hpx::parallel::util::projection_identity>
RandomIt nth_element(RandomIt first, RandomIt nth, Sent last, Pred &&pred = Pred(), Proj &&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

- **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`.

Returns The `nth_element` algorithm returns `RandomIt`. The `nth_element` algorithm returns an iterator equal to `last`.

```
template<typename ExPolicy, typename RandomIt, typename Sent, typename Pred =
hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result_t<ExPolicy, RandomIt> nth_element(ExPolicy &&policy,
RandomIt first, RandomIt nth, Sent last, Pred &&pred = Pred(), Proj &&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.

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.

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`.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
hpx::traits::range_iterator_t<Rng> nth_element(Rng &&rng, hpx::traits::range_iterator_t<Rng> nth,
                                                Pred &&pred = Pred(), Proj &&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

- **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`.

Returns The `nth_element` algorithm returns `hpx::traits::range_iterator_t<Rng>`. The `nth_element` algorithm returns an iterator equal to `last`.

```
template<typename ExPolicy, typename Rng, typename Pred = hpx::parallel::v1::detail::less, typename
Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result_t<ExPolicy, hpx::traits::range_iterator_t<Rng>> nth_element(ExPolicy
&&pol-
icy,
Rng
&&rng,
hpx::traits::range_ite
nth,
Pred
&&pred
=
Pred(),
Proj
&&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.

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.

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`.

Returns 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`.

hpx/parallel/container_algorithms/partial_sort.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename RandomIt, typename Sent, typename Comp = ranges::less, typename Proj = parallel::util::projection_identity>
RandomIt partial_sort(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 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.

Returns 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 = ranges::less,
         typename Proj = parallel::util::projection_identity>
parallel::util::algorithm_result<ExPolicy, RandomIt>::type 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.

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.

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.

Returns 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 = ranges::less, typename Proj = parallel::util::projection_identity>
hpx::traits::range_iterator_t<Rng> partial_sort(Rng &&rng, hpx::traits::range_iterator_t<Rng>
                                                middle, 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

- **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 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.

Returns The *partial_sort* algorithm returns *typename hpx::traits::range_iterator_t<Rng>*. It returns *last*.

```
template<typename ExPolicy, typename Rng, typename Comp = ranges::less, typename Proj = parallel::util::projection_identity>
```

```
parallel::util::detail::algorithm_result_t<ExPolicy, hpx::traits::range_iterator_t<Rng>> partial_sort(ExPolicy
    &&policy,
    Rng
    &&rng,
    hpx::traits::range_i
    midle,
    Comp
    &&comp
    =
    Comp(),
    Proj
    &&proj
    =
    Proj())
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

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.

Note: Complexity: O(Nlog(N)), where N = std::distance(first, 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 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.

Returns The `partial_sort` algorithm returns a `hpx::future<typename hpx::traits::range_iterator_t<Rng>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `typename hpx::traits::range_iterator_t<Rng>` otherwise. It returns `last`.

hpx/parallel/container_algorithms/partial_sort_copy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename InIter, typename Sent1, typename RandIter, typename Sent2, typename Comp = ranges::less, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
partial_sort_copy_result<InIter, RandIter> partial_sort_copy(InIter first, Sent1 last, RandIter r_first, Sent2 r_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 $[r_first, r_last)$. At most $r_last - r_first$ of the elements are placed sorted to the range $[r_first, r_first + n]$ where n is the number of elements to sort ($n = \min(last - first, r_last - r_first)$).

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 $D = \text{std::distance}(r_first, r_last)$ 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`.
- **Proj1** – 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.
- **r_first** – Refers to the beginning of the destination range.
- **r_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.

Returns 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 Comp = ranges::less, typename Proj1 = parallel::util::projection_identity, typename
Proj2 = parallel::util::projection_identity>
```

```
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 $[r_first, r_last)$. At most $r_last - r_first$ of the elements are placed sorted to the range $[r_first, r_first + n)$ where n is the number of elements to sort ($n = \min(last - first, r_last - r_first)$).

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.

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.

Note: Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ and $D = \text{std}::\text{distance}(\text{r_first}, \text{r_last})$ 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`.
- **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.
- **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.
- **r_first** – Refers to the beginning of the destination range.
- **r_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.

Returns 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 Rng1, typename Rng2, typename Comp = ranges::less, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
```

`partial_sort_copy_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>> partial_sort_copy(`

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [r_first, r_last). At most r_last - r_first of the elements are placed sorted to the range [r_first, r_first + n) where n is the number of elements to sort (n = min(last - first, r_last - r_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 $D = \text{std}::\text{distance}(\text{r_first}, \text{r_last})$ 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.

Returns 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 Rng1, typename Rng2, typename Comp = ranges::less,
typename Proj1 = parallel::util::projection_identity, typename Proj2 =
parallel::util::projection_identity>
parallel::util::detail::algorithm_result_t<ExPolicy, partial_sort_copy_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits
```

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [r_first, r_last). At most r_last - r_first of the elements are placed sorted to the range [r_first, r_first + n) where n is the number of elements to sort (n = min(last - first, r_last - r_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.

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.

Note: Complexity: $O(N \log(\min(D, N)))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ and $D = \text{std}::\text{distance}(\text{r_first}, \text{r_last})$ 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.

Returns 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}.

hpx/parallel/container_algorithms/partition.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename Rng, typename Pred, typename Proj = parallel::util::projection_identity>
 subrange_t<hpx::traits::range_iterator_t<Rng>> partition(Rng &&rng, Pred &&pred, Proj &&proj = Proj())
```

Reorders the elements in the range *rng* in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked 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.

Returns 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 =  
parallel::util::projection_identity>  
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<hpx::traits::range_iterator_t<Rng>>>::type partition(ExPo  
&&po  
icy,  
Rng  
&&rn  
Pred  
&&pr  
Proj  
&&pr  
=  
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.

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.

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.

Returns 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 `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 FwdIter, typename Sent, typename Pred, typename Proj =  
    parallel::util::projection_identity>  
subrange_t<FwdIter> partition(FwdIter first, Sent last, Pred &&pred, Proj &&proj = Proj())  
Reorders the elements in the range [first, last) in such a way that all elements for which the predicate  
pred returns true precede the elements for which the predicate pred returns false. Relative order of the  
elements is not preserved.
```

The assignments in the parallel *partition* algorithm invoked 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.

Returns 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 = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter>>::type partition(ExPolicy
    &&policy,
    FwdIter first,
    Sent last,
    Pred
    &&pred,
    Proj &&proj
    = Proj())
```

Reorders the elements in the range $[first, last)$ in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

Note: Complexity: At most $2 * (last - first)$ swaps. Exactly $last - first$ applications of the predicate and projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **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.

Returns 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 = parallel::util::projection\_identity>
subrange_t<hpx::traits::range\_iterator\_t<Rng>> stable_partition(Rng &&rng, Pred &&pred, Proj &&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

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an bidirectional 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** – 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.

Returns 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 Pred, typename Proj =  
parallel::util::projection_identity>  
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<hpx::traits::range_iterator_t<Rng>>>::type stable_partition
```

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.

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.

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*.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an bidirectional 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** – 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.

Returns 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 BidirIter, typename Sent, typename Pred, typename Proj =  
    parallel::util::projection_identity>  
subrange_t<BidirIter> stable_partition(BidirIter first, Sent last, Pred &&pred, Proj &&proj =  
    Proj())
```

Permutes the elements in the range `[first, last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first, i)` `INVOKE(f, INVOKE(proj, *j)) != false`, and for every iterator `k` in the range `[i, last)`, `INVOKE(f, INVOKE(proj, *k)) == false`

The invocations of `f` in the parallel `stable_partition` algorithm invoked 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`.
- **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** – 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.

Returns 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 Pred, typename Proj  
= parallel::util::projection_identity>  
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<BidirIter>>::type stable_partition(ExPolicy  
&&policy,  
BidirIter  
first,  
Sent  
last,  
Pred  
&&pred,  
Proj  
&&proj  
=  
Proj())
```

Permutes the elements in the range [*first*, *last*) such that there exists an iterator *i* such that for every iterator *j* in the range [*first*, *i*) *INVOKE*(*f*, *INVOKE*(*proj*, **j*)) != false, and for every iterator *k* in the range [*i*, *last*), *INVOKE*(*f*, *INVOKE*(*proj*, **k*)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

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.

Note: Complexity: At most (*last* - *first*) * log(*last* - *first*) swaps, but only linear number of swaps if there is enough extra memory Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for *BidirIter*.
- **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** – 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.

Returns 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 OutIter2, typename OutIter3, typename Pred, typename Proj = parallel::util::projection_identity>
partition_copy_result<hpx::traits::range_iterator_t<Rng>, OutIter2, OutIter3> partition_copy(Rng &&rng,
                                                                                         OutIter2 dest_true,
                                                                                         OutIter3 dest_false,
                                                                                         Pred &&pred,
                                                                                         Proj &&proj
                                                                                         = Proj())
```

Copies the elements in the range `rng`, to two different ranges depending on the value returned by the predicate `pred`. The elements, that satisfy the predicate `pred` are copied to the range beginning at `dest_true`. The rest of the elements are copied to the range beginning at `dest_false`. The order of the elements is preserved.

The assignments in the parallel `partition_copy` algorithm invoked 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.
- **OutIter2** – 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.
- **OutIter3** – 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 require-

ments 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.

Returns The *partition_copy* algorithm returns a partitioning *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 = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>> partition_copy(ExPolicy&& policy, FwdIter2 first, FwdIter2 last, FwdIter3 dest_true, FwdIter3 dest_false, Pred pred, Proj proj = parallel::util::projection_identity{})
```

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.

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.

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.

Returns 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.

template<typename **InIter**, typename **Sent**, typename **OutIter2**, typename **OutIter3**, typename **Pred**, typename **Proj** = *parallel::util::projection_identity*>

```
partition_copy_result<InIter, OutIter2, OutIter3> partition_copy(InIter first, Sent last, OutIter2  
dest_true, OutIter3 dest_false, Pred  
&&pred, Proj &&proj = Proj())
```

Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked 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

- **InIter** – 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.
- **OutIter2** – 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.
- **OutIter3** – 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.

Returns 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 FwdIter, typename Sent, typename OutIter2, typename OutIter3, typename Pred, typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, partition_copy_result<FwdIter, OutIter2, OutIter3>>::type partition_copy
```

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.

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.

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.
- **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.
- **OutIter2** – 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.
- **OutIter3** – 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 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.

Returns 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.

hpx/parallel/container_algorithms/reduce.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename F, typename T = typename std::iterator_traits<FwdIter>::value_type>
hpx::parallel::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.

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.

Note: Complexity: $O(\text{last} - \text{first})$ applications of the predicate *f*.

Note: GENERALIZED_SUM(*op*, *a*1, ..., *a*N) is defined as follows:

- *a*1 when N is 1
- *op*(GENERALIZED_SUM(*b*1, ..., *b*K), GENERALIZED_SUM(*b*M, ..., *b*N)), where:
 - *b*1, ..., *b*N may be any permutation of *a*1, ..., *a*N and
 - $1 < K+1 = M \leq N$.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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.

Returns 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).

```
template<typename ExPolicy, typename Rng, typename F, typename T = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
hpx::parallel::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.

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.

Note: Complexity: $O(last - first)$ applications of the predicate *f*.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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).

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename T = typename std::iterator_traits<FwdIter>::value_type>
```

`hpx::parallel::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.

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.

Note: Complexity: O(*last - first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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).

```
template<typename ExPolicy, typename Rng, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
hpx::parallel::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.

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.

Note: Complexity: $O(\text{last} - \text{first})$ applications of the operator`+()`.

Note: `GENERALIZED_SUM(+, a1, ..., aN)` is defined as follows:

- $a1$ when N is 1
 - $\text{op}(\text{GENERALIZED_SUM}(+, b1, \dots, bK), \text{GENERALIZED_SUM}(+, bM, \dots, bN))$, where:
 - $b1, \dots, bN$ may be any permutation of $a1, \dots, aN$ and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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.

Returns 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].

```
template<typename ExPolicy, typename FwdIter, typename Sent>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIter>::value_type>::type reduce(E
```

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.

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.

Note: Complexity: $O(last - first)$ applications of the operator $+$ ().

Note: The type of the initial value (and the result type) T is determined from the *value_type* of the used *FwdIterB*.

Note: GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
 - op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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).

template<typename **ExPolicy**, typename **Rng**>

`hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>`

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.

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.

Note: Complexity: $O(\text{last} - \text{first})$ applications of the operator+().

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
 - $\text{op}(\text{GENERALIZED_SUM}(+, b1, \dots, bK), \text{GENERALIZED_SUM}(+, bM, \dots, bN))$, where:
 - $b1, \dots, bN$ may be any permutation of $a1, \dots, aN$ and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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.

Returns 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).

```
template<typename FwdIter, typename Sent, typename F, typename T = typename  
std::iterator_traits<FwdIter>::value_type>  
T reduce(FwdIter first, Sent last, T init, F &&  
Returns GENERALIZED_SUM(f, init, *first, ..., *(first + (last - first) - 1)).
```

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: $O(\text{last} - \text{first})$ applications of the predicate f .

Note: `GENERALIZED_SUM(op, a1, ..., aN)` is defined as follows:

- $a1$ when N is 1
 - $\text{op}(\text{GENERALIZED_SUM}(op, b1, \dots, bK), \text{GENERALIZED_SUM}(op, bM, \dots, bN))$, where:
 - $b1, \dots, bN$ may be any permutation of $a1, \dots, aN$ and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **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

- **first** – Refers to the beginning of the sequence 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.

Returns The *reduce* algorithm returns *T*. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

```
template<typename Rng, typename F, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
T reduce(Rng &&rng, T init, F &&f)
```

Returns GENERALIZED_SUM(f, init, *first, ..., *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: $O(last - first)$ applications of the predicate *f*.

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, ..., *b*_N may be any permutation of *a*₁, ..., *a*_N and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **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

- **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.

Returns The `reduce` algorithm returns `T`. The `reduce` algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

```
template<typename FwdIter, typename Sent, typename T = typename  
std::iterator_traits<FwdIter>::value_type>  
T reduce(FwdIter first, Sent last, T init)
```

Returns `GENERALIZED_SUM(+, init, *first, ..., *(first + (last - first) - 1))`.

The difference between `reduce` and `accumulate` is that the behavior of `reduce` may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: $O(last - first)$ applications of the operator `+`.

Note: `GENERALIZED_SUM(+, a1, ..., aN)` 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 Parameters

- **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

- **first** – Refers to the beginning of the sequence 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.

Returns The `reduce` algorithm returns `T`. The `reduce` algorithm returns the result of the generalized sum (applying operator `+`) over the elements given by the input range [first, last).

```
template<typename Rng, typename T = typename  
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>  
T reduce(Rng &&rng, T init)
```

Returns `GENERALIZED_SUM(+, init, *first, ..., *(first + (last - first) - 1))`.

The difference between `reduce` and `accumulate` is that the behavior of `reduce` may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: $O(last - first)$ applications of the operator `+`.

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 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 used as initial (and intermediate) values (deduced).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns *T*. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

```
template<typename FwdIter, typename Sent>
std::iterator_traits<FwdIter>::value_type reduce(FwdIter first, Sent last)
    Returns GENERALIZED_SUM(+, T(), *first, ..., *(first + (last - first) - 1)).
```

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: $O(last - first)$ applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the *value_type* of the used *FwdIterB*.

Note: GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
 - op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **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

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns *T* (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).

```
template<typename Rng>
```

```
std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_type>::value_type reduce(Rng  
&&rng)
```

Returns GENERALIZED_SUM(+, T(), *first, ..., *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of *reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the *value_type* of the used *FwdIterB*.

Note: GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
 - b1, ..., bN may be any permutation of a1, ..., aN and
 - 1 < K+1 = M <= N.

Template 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 sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns *T* (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].

hpx/parallel/container_algorithms/remove.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename Iter, typename Sent, typename Pred, typename Proj =  
hpx::parallel::util::projection_identity>  
subrange_t<Iter, Sent> remove_if(Iter 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

- **Iter** – 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.
- **sent** – 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.

Returns 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 Rng, typename Pred, typename Proj = hpx::parallel::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.

Returns 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 FwdIter, typename Sent, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
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.

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.

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.
- **sent** – 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.

Returns 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 ExPolicy, typename Rng, typename Pred, typename Proj =
hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<typename hpx::traits::range_iterator<Rng>::type>>::type remove_if(
```

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.

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.

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.

Returns 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.

```
template<typename Iter, typename Sent, typename Proj = hpx::parallel::util::projection_identity, typename T = typename hpx::parallel::traits::projected<Iter, Proj>::value_type>
subrange_t<Iter, Sent> remove(Iter 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

- **Iter** – 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.

Returns 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 Rng, typename Proj = hpx::parallel::util::projection_identity, typename T =
typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>, Proj>::value_type>
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.

Returns 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 FwdIter, typename Sent, typename Proj =
hpx::parallel::util::projection_identity, typename T = typename hpx::parallel::traits::projected<FwdIter,
Proj>::value_type>
```

```
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter, Sent>>::type remove(ExPolicy
    &&policy,
    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.

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.

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.

Returns 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 ExPolicy, typename Rng, typename Proj = hpx::parallel::util::projection_identity,
        typename T = typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>,
        Proj>::value_type>
```

`parallel::util::detail::algorithm_result<ExPolicy, subrange_t<typename hpx::traits::range_iterator<Rng>::type>>::type remove`

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 invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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.

Returns 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.

hpx/parallel/container_algorithms/remove_copy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **ranges**

hpx/parallel/container_algorithms/replace.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **ranges**

Functions

```
template<typename Iter, typename Sent, typename Pred, typename Proj =
    hpx::parallel::util::projection_identity, typename T = typename hpx::parallel::traits::projected<Iter,
    Proj>::value_type>
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`

The assignments in the parallel *replace_if* 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 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

- **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.

- **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 `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.

Returns The `replace_if` algorithm returns `Iter`. It returns `last`.

```
template<typename Rng, typename Pred, typename Proj = hpx::parallel::util::projection_identity,
typename T = typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>,
Proj>::value_type>
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.

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 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.

Returns 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 Proj = hpx::parallel::util::projection_identity, typename T = typename hpx::parallel::traits::projected<Iter, Proj>::value_type>
parallel::util::detail::algorithm_result<ExPolicy, Iter>::type replace_if(ExPolicy &&policy, Iter first,
                                                               Sent sent, 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*

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.

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.

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.
- **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.
- **sent** – 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:

<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`.

- **`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.

Returns 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 Proj =
hpx::parallel::util::projection_identity, typename T = typename
hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>, Proj>::value_type>
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type replace_if(ExP
&&
icy,
Rng
&&
Pre
&&
T
con
&ne
Pro
&&
=
Pro
```

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`

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.

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.

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 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 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.

Returns 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 Iter, typename Sent, typename Proj = hpx::parallel::util::projection_identity, typename T1 = typename hpx::parallel::traits::projected<Iter, Proj>::value_type, typename T2 = T1>
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: `INVOKED(proj, *i) == 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

- **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.
- **sent** – 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.

Returns The *replace* algorithm returns an *Iter*.

```
template<typename Rng, typename Proj = hpx::parallel::util::projection_identity, typename T1 =
typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>, Proj>::value_type,
typename T2 = T1>
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: $\text{INVOKED}(\text{proj}, *i) == \text{old_value}$

The assignments in the parallel *replace* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *util::end(rng)* - *util::begin(rng)* 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.
- **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.

Returns The *replace* algorithm returns an *hpx::traits::range_iterator<Rng>::type*.

```
template<typename ExPolicy, typename Iter, typename Sent, typename Proj =
hpx::parallel::util::projection_identity, typename T1 = typename hpx::parallel::traits::projected<Iter,
Proj>::value_type, typename T2 = T1>
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: $\text{INVOKED}(\text{proj}, *i) == \text{old_value}$

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.

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.

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 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.
- **sent** – 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.

Returns 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 Proj = hpx::parallel::util::projection_identity,
typename T1 = typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>,
Proj>::value_type, typename T2 = T1>
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type replace(ExPolicy
&&policy,
Rng
&&rng,
T1
const
&old_v
T2
const
&new_v
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`

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.

Note: Complexity: Performs exactly `util::end(rng) - util::begin(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 a forward iterator.
- **T1** – The type of the old value to replace (deduced).
- **T2** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate is invoked. The assignments in the parallel `replace` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

Returns 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 InIter, typename Sent, typename OutIter, typename Pred, typename T =
typename std::iterator_traits<OutIter>::value_type, typename Proj =
hpx::parallel::util::projection_identity>
replace_copy_if_result<InIter, OutIter> replace_copy_if(InIter first, Sent sent, OutIter dest, Pred
&&pred, T const &new_value, Proj
&&proj = Proj())
```

Copies the all elements from the range $[first, sent)$ to another range beginning at $dest$ replacing all elements satisfying a specific criteria with new_value .

Effects: Assigns to every iterator it in the range $[result, result + (sent - first))$ either new_value or $*(first + (it - result))$ depending on whether the following corresponding condition holds: `INVOKE(f, INVOKE(proj, *(first + (i - result)))) != false`

The assignments in the parallel `replace_copy_if` algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly $sent - first$ applications of the predicate.

Template Parameters

- **InIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for `InIter`.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (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 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.

Returns 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  
std::iterator_traits<OutIter>::value_type, typename Proj = hpx::parallel::util::projection_identity>  
replace_copy_if_result<typename hpx::traits::range_iterator<Rng>::type, OutIter> replace_copy_if(Rng  
&&rng,  
Out-  
Iter  
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 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.

Returns The `replace_copy_if` algorithm returns an `in_out_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>`. The `replace_copy_if` algorithm returns the input iterator `last` and the output iterator to the element in the destination range, one past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename Pred, typename T = typename std::iterator_traits<FwdIter2>::value_type, typename Proj = hpx::parallel::util::projection_identity>
```

`parallel::util::detail::algorithm_result<ExPolicy, replace_copy_if_result<FwdIter1, FwdIter2>>::type replace_copy_if()`

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.

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.

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.

Returns 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 std::iterator_traits<FwdIter>::value_type, typename Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy>, replace_copy_if_result<typename hpx::traits::range_iterator<Rng>::type, I>
```

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: $\text{INVOKE}(f, \text{INVOKED}(proj, *(first + (i - result)))) \neq \text{false}$

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.

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.

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.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of `equal` requires `Pred` to meet the requirements of `CopyConstructible`. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate which returns `true` for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
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.

Returns 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.

```
template<typename InIter, typename Sent, typename OutIter, typename Proj =
hpx::parallel::util::projection_identity, typename T1 = typename hpx::parallel::traits::projected<InIter, Proj>::value_type, typename T2 = T1>
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

- **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 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.

Returns 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 Proj = hpx::parallel::util::projection_identity,
typename T1 = typename hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>,
Proj>::value_type, typename T2 = T1>
replace_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter> replace_copy(Rng
&&rng,
OutIter
dest, T1
const
&old_value,
T2
const
&new_value,
Proj
&&proj
=
Proj()
```

Copies the all elements from the range *rbg* to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator *it* in the range $[result, result + (util::end(rng) - util::begin(rng))]$ either *new_value* or $*(first + (it - result))$ 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.

Returns 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 Proj = hpx::parallel::util::projection_identity, typename T1 = typename hpx::parallel::traits::projected<FwdIter1, Proj>::value_type, typename T2 = T1> parallel::util::detail::algorithm_result<ExPolicy, replace_copy_result<FwdIter1, FwdIter2>>::type replace_copy(ExPolicy &&policy, FwdIter1 first, Sent sent, FwdIter2 dest, T1 const &old_value, T2 const &new_value, Proj &&projection = 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: `INVOKED(proj, *(first + (i - result))) == old_value`

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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.

Returns 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 Proj =
    hpx::parallel::util::projection_identity, typename T1 = typename
    hpx::parallel::traits::projected<hpx::traits::range_iterator_t<Rng>, Proj>::value_type, typename T2 =
    T1>
```

parallel::util::detail::algorithm_result<ExPolicy, replace_copy_result<typename hpx::traits::range_iterator<Rng>::type, FwdIter, T1, T2, 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: $\text{INVOKE}(\text{proj}, *(\text{first} + (\text{it} - \text{result}))) == \text{old_value}$

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.

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.

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.

Returns 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.

hpx/parallel/container_algorithms/reverse.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **Iter**, typename **Sent**>
`Iter` **reverse**(`Iter` first, `Sent` sent)

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 $i < (\text{last-first})/2$.

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.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns 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 $i < (\text{last-first})/2$.

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.

Returns 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
sent)
```

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 $i < (\text{last-first})/2$.

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.

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.

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.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns 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 $i < (\text{last-first})/2$.

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.

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.

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.

Returns 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 $*(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [first, last) and [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.

Returns 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**>
reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter> reverse_copy(Rng &&rng, OutIter result)

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Copies the elements from the range [first, last) to another range beginning at result in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment $*(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i)$ once for each non-negative $i < (\text{last} - \text{first})$. If the source and destination ranges (that is, [first, last) and [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.
- **OutIter** – 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.

Returns The *reverse_copy* algorithm returns a *ranges::reverse_copy_result< typename hpx::traits::range_iterator<Rng>::type, OutIter>*. The *reverse_copy* algorithm returns an object equal to {last, result + N} where N = last - first

```
template<typename ExPolicy, typename Iter, typename Sent, typename FwdIter>
parallel::util::detail::algorithm_result<ExPolicy, reverse_copy_result<Iter, FwdIter>>::type reverse_copy(ExPolicy &&policy,
&&pol-  
icy,  
Iter  
first,  
Sent  
last,  
FwdIter  
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 $*(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i)$ 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.

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.

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.
- **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.
- **result** – Refers to the begin of the destination range.

Returns The `reverse_copy` algorithm returns a `hpx::future<reverse_copy_result<Iter, FwdIter>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `reverse_copy_result<Iter, FwdIter>` 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>
parallel::util::detail::algorithm_result<ExPolicy, reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>> reverse_copy(ExPolicy policy, InputIterator first, InputIterator last, OutputIterator result);
```

Uses `rng` as the source range, as if using `util::begin(rng)` as `first` and `ranges::end(rng)` as `last`. Copies the elements from the range `[first, last)` to another range beginning at `result` in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment `*(result + (last - first) - 1 - i) = *(first + i)` once for each non-negative `i < (last - first)`. If the source and destination ranges (that is, `[first, last)` and `[result, result+(last-first))` respectively) overlap, the behavior is undefined.

The assignments in the parallel `reverse_copy` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

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.

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.

- **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.
- **result** – Refers to the begin of the destination range.

Returns The *reverse_copy* algorithm returns a *hpx::future<ranges::reverse_copy_result< typename hpx::traits::range_iterator<Rng>::type, OutIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::reverse_copy_result< typename hpx::traits::range_iterator<Rng>::type, OutIter>* otherwise. The *reverse_copy* algorithm returns an object equal to {last, result + N} where N = last - first

[hpx/parallel/container_algorithms/rotate.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

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.

Returns 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.

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: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *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 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.

Returns The *rotate* algorithm returns a *hpx::future<subrange_t<FwdIter, Sent>>* if the execution policy is of type *parallel_task_policy* and returns a *subrange_t<FwdIter, Sent>* otherwise. The *rotate* algorithm returns the iterator equal to pair(*first* + (*last* - *middle*), *last*).

```
template<typename Rng>
subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> rotate(Rng
&&rng,
hpx::traits::range_iterator_t<Rng> middle)
```

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [*first*, *last*) in

such a way that the element middle becomes the first element of the new range and middle - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

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.

Returns 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**>

parallel::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.

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: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *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 executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.

Returns The *rotate* algorithm returns a `hpx::future <sub-range_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.
- **dest_first** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first, last).

Returns The *rotate_copy* algorithm returns a `rotate_copy_result<FwdIter, OutIter>`. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2>
```

```
parallel::util::detail::algorithm_result<ExPolicy, rotate_copy_result<FwdIter1, FwdIter2>>::type rotate_copy(ExPolicy  
    &&policy,  
    FwdIter1  
    first,  
    FwdIter1  
    middle,  
    Sent  
    last,  
    FwdIter2  
    dest_first)
```

Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *middle* becomes the first element of the new range and *middle* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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.
- **Sent** – The type of the end iterators used (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.
- **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.
- **dest_first** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first, last).

Returns The *rotate_copy* algorithm returns areturns 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** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first, last).

Returns 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 ro
```

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.

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.

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** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first,last).

Returns 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.

hpx/parallel/container_algorithms/search.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename FwdIter, typename Sent, typename FwdIter2, typename Sent2, typename Pred =  
hpx::ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 =  
parallel::util::projection_identity>  
FwdIter search(FwdIter first, Sent last, FwdIter2 s_first, Sent2 s_last, Pred &&op = Pred(), Proj1  
&&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most ($S \cdot N$) 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.
- **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.

Returns 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 Pred = hpx::ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
```

```
parallel::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.

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.

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.

Returns 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 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 $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **Rng1** – The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires `Pred` to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj1** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of `Rng1`.
- **Proj2** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of `Rng2`.

Parameters

- **rng1** – Refers to the sequence of elements the algorithm will be examining.
- **rng2** – Refers to the sequence of elements the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
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.

Returns 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 Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =  
hpx::parallel::util::projection_identity>  
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type search(Ex  
&&  
icy  
Rn  
&&  
Rn  
&&  
Pre  
&&  
=  
Pre  
Pre  
&&  
=  
Pre  
Pre  
&&  
=  
Pre  
Pre  
&&  
=  
Pre
```

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.

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.

Note: Complexity: at most $(S*N)$ comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it

executes the assignments.

- **Rng1** – The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj1** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of *Rng1*.
- **Proj2** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of *Rng2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the sequence of elements the algorithm will be examining.
- **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.

Returns 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 Proj1 = parallel::util::projection_identity, typename Proj2 =
parallel::util::projection_identity>
FwdIter search_n(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<>
- **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.
- **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.

Returns 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::ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type search_n(ExPolicy &&policy,
    FwdIter first,
    std::size_t count,
    FwdIter2 s_first, Sent2
    s_last, Pred &&op =
    Pred(), Proj1 &&proj1
    = Proj1(), Proj2
    &&proj2 = Proj2())
```

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

Note: Complexity: at most $(S*N)$ comparisons where $S = \text{distance}(\text{s_first}, \text{s_last})$ 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<>`
- **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.
- **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.

Returns 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::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity> hpx::traits::range_iterator<Rng1>::type search_n(Rng1 &&rng1, std::size_t count, Rng2 &&rng2, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Searches the range [first, last) for any elements in the range [s_first, s_last). 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 ($S \cdot N$) 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.
- **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.

Returns 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 Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =
    hpx::parallel::util::projection_identity>
hpx::parallel::util::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type search_n(
```

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.

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.

Note: Complexity: at most (S^*N) comparisons where $S = \text{distance}(s_first, s_last)$ and $N = \text{distance}(first, last)$.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`

- **Proj1** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of *Rng1*.
- **Proj2** – The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of *Rng2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the sequence of elements the algorithm will be examining.
- **count** – The number of elements to apply the algorithm on.
- **rng2** – Refers to the sequence of elements the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

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

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

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

Returns 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.

hpx/parallel/container_algorithms/set_difference.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,
         typename Iter3, typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
         hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, set_difference_result<Iter1, Iter3>>::type set_difference(ExPolicy
&&policy,
Iter1
first1,
Sent1
last1,
Iter2
first2,
Sent2
last2,
Iter3
dest,
Pred
&&op
=
Pred(),
Proj1
&&proj
=
Proj1(),
Proj2
&&proj
=
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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, set_difference_result<typename hpx::traits::range_iterator<Rng1>::ty`

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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.

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3,
typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
set_difference_result<Iter1, Iter3> set_difference(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.

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.

Template Parameters

- **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

- **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.

Returns The *set_difference* algorithm returns *ranges::set_difference_result<Iter1, Iter3>*. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename Rng1, typename Rng2, typename Iter3, typename Pred =
    hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename
    Proj2 = hpx::parallel::util::projection_identity>
```

```
set_difference_result<typename hpx::traits::range_iterator<Rng1>::type, Iter3> set_difference(Rng1
    &&rng1,
    Rng2
    &&rng2,
    Iter3
    dest,
    Pred
    &&op
    =
    Pred(),
    Proj1
    &&proj1
    =
    Proj1(),
    Proj2
    &&proj2
    =
    Proj2())
```

Constructs a sorted range beginning at *dest* consisting of all elements present in the range [*first1*, *last1*] and not present in the range [*first2*, *last2*). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [*first1*, *last1*] and *n* times in [*first2*, *last2*), it will be copied to *dest* exactly std::max(*m-n*, 0) times. The resulting range cannot overlap with either of the input ranges.

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.

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.
- **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

- **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.

Returns The `set_difference` algorithm returns `ranges::set_difference_result<Iter1, Iter3>`. 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.

hpx/parallel/container_algorithms/set_intersection.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,
typename Iter3, typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

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

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.

- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, set_intersection_result<typename hpx::traits::range_iterator<Rng1>::`

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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.

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 $\text{std}::\text{min}(m, n)$ 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.

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.

Template Parameters

- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *util::projection_identity*

Parameters

- **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.

Returns The *set_intersection* algorithm returns *ranges::set_intersection_result<Iter1, Iter2, Iter3>*. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename Rng1, typename Rng2, typename Iter3, typename Pred =  
hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename  
Proj2 = hpx::parallel::util::projection_identity>
```

`set_intersection_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type,`

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.

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.

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.
- **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

- **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.

Returns The `set_intersection` algorithm returns `ranges::set_intersection_result<Iter1, Iter2, Iter3>`. 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.

hpx/parallel/container_algorithms/set_symmetric_difference.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,
typename Iter3, typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, set_symmetric_difference_result<Iter1, Iter2, Iter3>>::type` **set_symmetric_difference**

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, set_symmetric_difference_result<typename hpx::traits::range_iterator`

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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 *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.

Returns 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.

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3,
typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
set_symmetric_difference_result<Iter1, Iter2, Iter3> set_symmetric_difference(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 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.

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.

Template Parameters

- **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

- **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.

Returns The `set_symmetric_difference` algorithm returns `ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>`. The `set_symmetric_difference` algorithm returns the output iterator to the element in the

destination range, one past the last element copied.

```
template<typename Rng1, typename Rng2, typename Iter3, typename Pred =  
    hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename  
    Proj2 = hpx::parallel::util::projection_identity>  
set_symmetric_difference_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type>
```

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.

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.

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.
- **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

- **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.

Returns The `set_symmetric_difference` algorithm returns
`ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>`. 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.

hpx/parallel/container_algorithms/set_union.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2,
        typename Iter3, typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, 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 $\text{std}::\text{max}(n-m, 0)$ elements will be copied from $[first2, last2)$ to *dest*, also preserving order.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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 = hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

`hpx::parallel::util::detail::algorithm_result<ExPolicy, set_union_result<typename hpx::traits::range_iterator<Rng1>::type, t`

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.

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*).

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.

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.

Template Parameters

- **ExPolicy** – The type of the execution 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.

Returns 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.

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3,
typename Pred = hpx::parallel::v1::detail::less, typename Proj1 =
hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
set_union_result<Iter1, Iter2, Iter3> tagFallbackInvoke(set_union_t, 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.

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.

Template Parameters

- **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

- **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.

Returns The *set_union* algorithm returns *ranges::set_union_result<Iter1, Iter2, Iter3>*. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

```
template<typename Rng1, typename Rng2, typename Iter3, typename Pred =  
hpx::parallel::v1::detail::less, typename Proj1 = hpx::parallel::util::projection_identity, typename  
Proj2 = hpx::parallel::util::projection_identity>
```

`set_union_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type, Iter3>`

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.

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.

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.
- **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 *Copy-Constructible*. 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

- **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.

Returns The `set_union` algorithm returns `ranges::set_union_result<Iter1, Iter2, Iter3>`. 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.

hpx/parallel/container_algorithms/shift_left.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename FwdIter, typename Sent, typename Size>
FwdIter shift_left(FwdIter first, Sent last, Size n)
```

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first]

- n), moves the element originally at position `first + n + i` to position `first + i`.

The assignment operations in the parallel `shift_left` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

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.

Returns 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>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.

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: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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.

Note: The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

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.

Returns 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.

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: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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.

hpx/parallel/container_algorithms/shift_right.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

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.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

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.

Returns 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 **Size**>

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.

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: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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 Size>  
hpx::traits::range_iterator_t<Rng> 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.

Note: The type of dereferenced `hpx::traits::range_iterator_t<Rng>` must meet the requirements of `MoveAssignable`.

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.

Returns 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.

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: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced `hpx::traits::range_iterator_t<Rng>` must meet the requirements of `MoveAssignable`.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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.

Returns 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 re-

turns `hpx::traits::range_iterator_t<Rng>` otherwise. The `shift_right` algorithm returns an iterator to the end of the resulting range.

hpx/parallel/container_algorithms/sort.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename RandomIt, typename Sent, typename Comp = ranges::less, typename Proj = parallel::util::projection_identity>
RandomIt sort(RandomIt first, Sent last, Comp &&comp = Comp(), Proj &&proj = Proj())
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object `comp` (defaults to using operator`<()`).

A sequence is sorted with respect to a comparator `comp` and a projection `proj` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, and `INVOKED(comp, INVOKED(proj, *(i + n)), INVOKED(proj, *i)) == false`.

`comp` has to induce a strict weak ordering on the values.

The assignments in the parallel `sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: $O(N \log(N))$, where $N = \text{detail}::\text{distance}(\text{first}, \text{last})$ 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).
- **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 `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.

Returns 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 = ranges::less,
typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, RandomIt>::type sort(ExPolicy &&policy, RandomIt
first, Sent last, Comp &&comp
= Comp(), Proj &&proj =
Proj())
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i* + *n* is a valid iterator pointing to an element of the sequence, and *INVOKE*(*comp*, *INVOKE*(*proj*, *(*i* + *n*)), *INVOKE*(*proj*, **i*)) == false.

comp has to induce a strict weak ordering on the values.

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.

Note: Complexity: O(Nlog(N)), where N = detail::distance(first, 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.
- **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.

Returns 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, Comp &&comp = Comp(), Proj &&proj = Proj())
```

Sorts the elements in the range *rng* in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object *comp* (defaults to using operator<())).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and *Invoke(comp, Invoke(proj, *(i + n)), Invoke(proj, *i)) == false*.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: $O(N \log(N))$, where $N = \text{std::distance}(\text{begin}(rng), \text{end}(rng))$ 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).
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the *Invoke* operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *sort* algorithm returns *typename hpx::traits::range_iterator<Rng>::type*. It returns *last*.

```
template<typename ExPolicy, typename Rng, typename Comp = ranges::less, typename Proj = parallel::util::projection_identity>
```

```
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type sort(ExPolicy
    &&policy,
    Rng
    &&rng,
    Comp
    &&comp =
    Comp(),
    Proj
    &&proj =
    Proj())
```

Sorts the elements in the range *rng* in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i* + *n* is a valid iterator pointing to an element of the sequence, and $\text{INVOKED}(\text{comp}, \text{INVOKED}(\text{proj}, *(\text{i} + \text{n})), \text{INVOKED}(\text{proj}, *i)) == \text{false}$.

comp has to induce a strict weak ordering on the values.

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.

Note: Complexity: $O(N\log(N))$, where $N = \text{std}::\text{distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$ 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 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.

Returns The *sort* algorithm returns a *hpx::future<typename hpx::traits::range_iterator_t<Rng>* 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*.

hpx/parallel/container_algorithms/stable_sort.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename RandomIt, typename Sent, typename Comp = ranges::less, typename Proj = parallel::util::projection_identity>
RandomIt stable_sort(RandomIt first, Sent last, Comp &&comp = Comp{}, Proj &&proj = Proj{})
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and *Invoke(comp, Invoke(proj, *(i + n)), Invoke(proj, *i)) == false*.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *stable_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(Nlog(N)), where N = std::distance(first, last) 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).
- **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.

Returns 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 = ranges::less,
typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, RandomIt>::type stable_sort(ExPolicy &&policy,
RandomIt first, Sent
last, Comp &&comp =
Comp(), Proj &&proj
= Proj())
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i* + *n* is a valid iterator pointing to an element of the sequence, and INVOKE(*comp*, INVOKE(*proj*, *(*i* + *n*)), INVOKE(*proj*, **i*)) == false.

comp has to induce a strict weak ordering on the values.

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.

Note: Complexity: O(Nlog(N)), where N = std::distance(first, 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.
- **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.

Returns 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 = ranges::less, typename Proj = parallel::util::projection_identity>
hpx::traits::range_iterator<Rng>::type stable_sort(Rng &&rng, Comp &&comp = Comp(), Proj &&proj = Proj())
```

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object *comp* (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator *i* pointing to the sequence and every non-negative integer *n* such that *i + n* is a valid iterator pointing to an element of the sequence, and *(INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false)*.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *stable_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(Nlog(N)), where N = std::distance(first, last) 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).
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the *INVOKE* operation applied to an object of type *Comp*, when contextually converted to *bool*, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that *comp* will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *stable_sort* algorithm returns *typename hpx::traits::range_iterator<Rng>::type*. It returns *last*.

```
template<typename ExPolicy, typename Rng, typename Comp = ranges::less, typename Proj = parallel::util::projection_identity>
```

parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type **stable_sort**(ExPolicy &policy, Rng &rng, Comp &comp, Proj &proj = util::projection_identity)

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 $\text{INVOKED}(\text{comp}, \text{INVOKED}(\text{proj}, *(i + n)), \text{INVOKED}(\text{proj}, *i)) == \text{false}$.

comp has to induce a strict weak ordering on the values.

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.

Note: Complexity: $O(N \log(N))$, where $N = \text{std}::\text{distance}(\text{first}, \text{last})$ 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 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.

Returns The *stable_sort* algorithm returns a *hpx::future<typename hpx::traits::range_iterator_t<Rng>* 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*.

hpx/parallel/container_algorithms/starts_with.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred = ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
bool starts_with(Iter1 first1, Sent1 last1, Iter2 first2, Sent2 last2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&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(N1, N2) applications of the predicate and both projections.

Template Parameters

- **Iter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *starts_with* algorithm returns *bool*. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename Pred = ranges::equal_to, typename Proj1 = parallel::util::projection_identity,
         typename Proj2 = parallel::util::projection_identity>
parallel::detail::algorithm_result<ExPolicy, bool>::type starts_with(ExPolicy &&policy,
                                                               FwdIter1 first1, Sent1 last1,
                                                               FwdIter2 first2, Sent2 last2,
                                                               Pred &&pred = Pred(),
                                                               Proj1 &&proj1 = Proj1(),
                                                               Proj2 &&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.

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.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **FwdIter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.

- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate is invoked.

Returns 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 = ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
bool starts_with(Rng1 &&rng1, Rng2 &&rng2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(),
                  Proj2 &&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 $\min(N_1, N_2)$ applications of the predicate and both projections.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to `util::projection_identity`
- **Proj2** – The type of an optional projection function for the destination range. This defaults to `util::projection_identity`

Parameters

- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the 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.

Returns 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 = ranges::equal_to,
        typename Proj1 = parallel::util::projection_identity, typename Proj2 =
        parallel::util::projection_identity>
```

```
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type starts_with(ExPolicy &&policy,
    Rng1 &&rng1, Rng2
    &&rng2, Pred
    &&pred = Pred(),
    Proj1 &&proj1 =
    Proj1(), Proj2
    &&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.

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.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns 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.

hpx/parallel/container_algorithms/swap_ranges.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

Returns 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.

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.

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.

Returns 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<typename hpx::traits::range_iterator_t<Rng1>, typename hpx::traits::range_iterator_t<Rng2>> swap_ran
```

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.

Returns 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<typename hpx::traits::range_iterator_t<Rng1>, typenam
```

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.

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.

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.

Returns 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`.

hpx/parallel/container_algorithms/transform.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename F, typename Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, ranges::unary_transform_result<FwdIter1, FwdIter2>>::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.

The invocations of f in the parallel `transform` algorithm invoked with an execution policy object of type `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: 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 a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for FwdIter1.
- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a 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.

Returns 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 Rng, typename FwdIter, typename F, typename Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, ranges::unary_transform_result<typename hpx::traits::range_iterator<Rng
```

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.

The invocations of f in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: 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.
- **FwdIter** – The type of the iterator representing the destination 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 *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.

Returns The *transform* algorithm returns a `hpx::future<ranges::unary_transform_result<range_iterator<Rng>::type>`

`FwdIter>>` if the execution policy is of type *parallel_task_policy* and returns `ranges::unary_transform_result<range_iterator<Rng>::type, FwdIter>` 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 FwdIter3, typename F, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```

parallel::util::detail::algorithm_result<ExPolicy, ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>>::type **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.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: 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 a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for *FwdIter1*.
- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for *FwdIter2*.
- **FwdIter3** – The type of the source iterators for the first range 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 *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.

Returns 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.

```
template<typename ExPolicy, typename Rng1, typename Rng2, typename FwdIter, typename F,  
typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 =  
hpx::parallel::util::projection_identity

```

parallel::util::detail::algorithm_result<ExPolicy, ranges::binary_transform_result<typename hpx::traits::range_iterator<Rng1>, typename hpx::traits::range_iterator<Rng2>, FwdIter> >

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.

The invocations of f in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly $\min(\text{last2}-\text{first2}, \text{last1}-\text{first1})$ applications of f

Note: The algorithm will invoke the binary predicate until it reaches the end of the shorter of the two given input sequences

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of f .
- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **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 *transform* requires F to meet the requirements of *CopyConstructible*.
- **Proj1** – The type of an optional projection function to be used for elements of the first sequence. This defaults to *util::projection_identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second sequence. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`. The types `Type1` and `Type2` must be such that objects of types `range_iterator<Rng1>::type` and `range_iterator<Rng2>::type` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively. The type `Ret` must be such that an object of type `FwdIter` 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.

Returns The `transform` algorithm returns a `hpx::future<ranges::binary_transform_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type, FwdIter> >` if the execution policy is of type `parallel_task_policy` and returns `ranges::binary_transform_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type, FwdIter>` otherwise. The `transform` algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

```
template<typename FwdIter1, typename Sent1, typename FwdIter2, typename F, typename Proj = hpx::parallel::util::projection_identity>
ranges::unary_transform_result<FwdIter1, FwdIter2> transform(FwdIter1 first, Sent1 last, FwdIter2 dest, F &&f, Proj &&proj = Proj())
```

Applies the given function `f` to the given range `rng` and stores the result in another range, beginning at `dest`.

Note: Complexity: Exactly `size(rng)` applications of `f`

Template Parameters

- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for `FwdIter1`.
- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a 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

- **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.

Returns The `transform` algorithm returns `ranges::unary_transform_result<FwdIter1, FwdIter2>`. 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 Rng, typename FwdIter, typename F, typename Proj = hpx::parallel::util::projection_identity> ranges::unary_transform_result<typename hpx::traits::range_iterator<Rng>::type, FwdIter> transform(Rng &&rng, FwdIter dest, F &&f, Proj &&proj = Proj())
```

Applies the given function `f` to the given range `rng` and stores the result in another range, beginning at `dest`.

Note: Complexity: Exactly `size(rng)` applications of `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.
- **FwdIter** – The type of the iterator representing the destination 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 `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.
- **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.

Returns The *transform* algorithm returns *ranges*::*unary_transform_result*<*range_iterator*<*Rng*>::*type*, *FwdIter*>. 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 FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename FwdIter3, typename F, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3> transform(FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, FwdIter3 dest, F &&f, Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
```

Applies the given function *f* to pairs of elements from two ranges: one defined by *rng* and the other beginning at *first2*, and stores the result in another range, beginning at *dest*.

Note: Complexity: Exactly size(*rng*) applications of *f*

Template Parameters

- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for *FwdIter1*.
- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for *FwdIter2*.
- **FwdIter3** – The type of the source iterators for the first range 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 *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

- **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.

Returns The `transform` algorithm returns `ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>`. The `transform` algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

```
template<typename Rng1, typename Rng2, typename FwdIter, typename F, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity> ranges::binary_transform_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type> transform(FwdIter first1, FwdIter last1, FwdIter first2, FwdIter last2, F f, Proj1 proj1 = hpx::parallel::util::projection_identity{}, Proj2 proj2 = hpx::parallel::util::projection_identity{});
```

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`.

Note: Complexity: Exactly $\min(\text{last2}-\text{first2}, \text{last1}-\text{first1})$ applications of `f`

Note: The algorithm will invoke the binary predicate until it reaches the end of the shorter of the two

given input sequences

Template Parameters

- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **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 *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

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last]. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`. The types *Type1* and *Type2* must be such that objects of types `range_iterator<Rng1>::type` and `range_iterator<Rng2>::type` can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter* 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.

Returns The *transform* algorithm returns `ranges::binary_transform_result<typename hpx::traits::range_iterator<Rng1>::type, typename hpx::traits::range_iterator<Rng2>::type, FwdIter>`. 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.

hpx/parallel/container_algorithms/transform_exclusive_scan.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename InIter, typename Sent, typename OutIter, typename BinOp, typename UnOp,
typename T = typename std::iterator_traits<InIter>::value_type>
transform_exclusive_scan_result<InIter, OutIter> transform_exclusive_scan(InIter first, Sent last,
OutIter dest, T init,
BinOp &&binary_op,
UnOp &&unary_op)
```

Assigns through each iterator i in $[result, result + (last - first))$ 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.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(*op*, a1, \dots , 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 $1 < K+1 = M \leq N$.
-

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).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.
- **init** – The initial value for the generalized sum.
- **binary_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.

- **unary_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 `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`.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename BinOp, typename UnOp, typename T = typename std::iterator_traits<FwdIter1>::value_type>
parallel::util::detail::algorithm_result<ExPolicy, transform_exclusive_result<FwdIter1, FwdIter2>>::type transform_excl
```

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.

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)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.
- **init** – The initial value for the generalized sum.
- **binary_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.

- **unary_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 *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*.

Returns 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.

```
template<typename Rng, typename O, typename BinOp, typename UnOp, typename T = typename
std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
transform_exclusive_scan_result<hpx::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), ..., 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.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

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 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).
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.

- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **binary_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.

- **unary_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 `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`.

Returns The `transform_exclusive_scan` algorithm returns a `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.

```
template<typename ExPolicy, typename Rng, typename O, typename BinOp, typename UnOp, typename  
T = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>  
parallel::util::detail::algorithm_result<ExPolicy, transform_exclusive_scan_result<hpx::traits::range_iterator_t<Rng>, O>>
```

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.

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)].

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.
- **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.
- **binary_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.

- **unary_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 *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*.

Returns 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.

hpx/parallel/container_algorithms/transform_inclusive_scan.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename InIter, typename Sent, typename OutIter, typename BinOp, typename UnOp>
transform_inclusive_scan_result<InIter, OutIter> transform_inclusive_scan(InIter first, Sent last,
                                                                     OutIter dest, BinOp
                                                                     &&binary_op, UnOp
                                                                     &&unary_op)
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM($op, conv(*first), \dots, conv(*(\text{first} + (i - result)))$).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

The behavior of *transform_inclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_N) is defined as:

- a_1 when N is 1
 - $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_M, \dots, a_N))$ where $1 < K+1 = M \leq N$.
-

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.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.
- **binary_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.

- **unary_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 `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`.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename BinOp, typename UnOpparallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_result<FwdIter1, FwdIter2>>::type transform_incl
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of GENERALIZED_NONCOMMUTATIVE_SUM($op, conv(*first), \dots, conv(*(first + (i - result)))$)).

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.

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)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.
- **binary_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.

- **unary_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 *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*.

Returns 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.

```
template<typename Rng, typename O, typename BinOp, typename UnOp>
transform_inclusive_scan_result<hpx::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($op, conv(*first), \dots, conv(*(first + (i - result)))$)).

The reduce operations in the parallel `transform_inclusive_scan` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges $[first, last)$ or $[result, result + (last - first))$.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates `op` and `conv`.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_N) is defined as:

- a_1 when N is 1
- $op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \dots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_M, \dots, a_N))$ where $1 < K+1 = M \leq N$.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.
- **dest** – Refers to the beginning of the destination range.
- **binary_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.

- **unary_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 `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`.

Returns The `transform_inclusive_scan` algorithm returns a `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.

```
template<typename ExPolicy, typename Rng, typename O, typename BinOp, typename UnOp>
parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_scan_result<hpx::traits::range_iterator_t<Rng>, O>>;
```

Assigns through each iterator i in $[result, result + (last - first))$ the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, 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.

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 $[result, result + (last - first))$.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates `op` and `conv`.

Note: `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \dots, 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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_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.

- **unary_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 `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`.

Returns 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.

```
template<typename InIter, typename Sent, typename OutIter, typename BinOp, typename UnOp,
typename T = typename std::iterator_traits<InIter>::value_type>
transform_inclusive_scan_result<InIter, OutIter> transform_inclusive_scan(InIter first, Sent 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(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.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of *transform_inclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

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 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.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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).

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.
- **binary_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.

- **unary_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 *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.

Returns 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.

```
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename
BinOp, typename UnOp, typename T = typename std::iterator_traits<FwdIter1>::value_type>
parallel::util::algorithm_result<ExPolicy, transform_inclusive_result<FwdIter1, FwdIter2>>::type transform_incl
```

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.

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 $[result, result + (last - first))$.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates `op` and `conv`.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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).

Parameters

- **policy** – The execution policy to use for 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.
- **binary_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.

- **unary_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 `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.

Returns 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.

```
template<typename Rng, typename O, typename BinOp, typename UnOp, typename T = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
transform_inclusive_scan_result<hpx::traits::range_iterator_t<Rng>, O> transform_inclusive_scan(Rng &&rng, O dest, BinOp &&binary_op, UnOp &&unary_op, T init)
```

Assigns through each iterator i in $[result, result + (last - first))$ 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.

Neither *conv* nor *op* shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

The behavior of *transform_inclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

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 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).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_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.

- **unary_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 *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.

Returns The *transform_inclusive_scan* algorithm returns a *transform_inclusive_scan_result<* traits::range_iterator_t<*Rng*>, *O**>*. The *trans-*

form_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.

```
template<typename ExPolicy, typename Rng, typename O, typename BinOp, typename UnOp, typename  
T = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>  
parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_scan_result<hpx::traits::range_iterator_t<Rng>, O>>;
```

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.

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)).

The behavior of *transform_inclusive_scan* may be non-deterministic for a non-associative predicate.

Note: Complexity: $O(last - first)$ applications of the predicates *op* and *conv*.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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.

- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – 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).

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.
- **binary_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.

- **unary_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 *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.

Returns 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.

hpx/parallel/container_algorithms/transform_reduce.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ExPolicy, typename Iter, typename Sent, typename T, typename Reduce, typename Convert>
hpx::parallel::util::detail::algorithm_result<ExPolicy, T>::type transform_reduce(ExPolicy &&policy, Iter first, Sent 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.

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.

Note: Complexity: $O(last - first)$ applications of the predicates *red_op* and *conv_op*.

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, ..., *b*N may be any permutation of *a*1, ..., *a*N and
 - $1 < K+1 = M \leq N$.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 Iter.
- **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 first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range 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 & <i>a</i> , const Type2 & <i>b</i>);

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`.

Returns 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).

template<typename **Iter**, typename **Sent**, typename **T**, typename **Reduce**, typename **Convert**>
T **transform_reduce**(**Iter** first, **Sent** 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 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.

Note: Complexity: $O(last - first)$ applications of the predicates `red_op` and `conv_op`.

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 Parameters

- **`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 `Iter`.
- **`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

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range 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*.

Returns The *transform_reduce* algorithm returns *T*. 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).

```
template<typename ExPolicy, typename Iter, typename Sent, typename Iter2, typename T>  
hpx::parallel::util::detail::algorithm_result<ExPolicy, T>::type transform_reduce(ExPolicy &&policy, Iter  
first, Sent last, Iter2  
first2, T init)
```

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.

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.

Note: Complexity: $O(last - first)$ applications of the predicates *red_op* and *conv_op*.

Note: `GENERALIZED_SUM(op, a1, ..., aN)` is defined as follows:

- $a1$ when N is 1
- $op(GENERALIZED_SUM(op, b1, \dots, bK), GENERALIZED_SUM(op, bM, \dots, bN))$, where:

-
- b_1, \dots, b_N may be any permutation of a_1, \dots, a_N and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (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 Iter.
- **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.
- **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 first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted 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.
- **init** – The initial value for the generalized sum.

Returns 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).

```
template<typename Iter, typename Sent, typename Iter2, typename T>
T transform_reduce(Iter first, Sent last, Iter2 first2, T init)
```

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), ..., conv_op(*(first + (last - first) - 1))).

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.

Note: Complexity: $O(last - first)$ applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(*op*, a_1, \dots, a_N) 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 Parameters

- **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 Iter.
- **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.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted 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.
- **init** – The initial value for the generalized sum.

Returns The *transform_reduce* algorithm returns *T*. 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].

```
template<typename ExPolicy, typename Iter, typename Sent, typename Iter2, typename T, typename Reduce, typename Convert>
hpx::parallel::util::algorithm_result<ExPolicy, T>::type transform_reduce(ExPolicy &&policy, Iter
first, Sent last, Iter2
first2, 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.

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.

Note: Complexity: O(*last - first*) applications of the predicates *red_op* and *conv_op*.

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, ..., *b*N may be any permutation of *a*1, ..., *a*N and

-
- $1 < K+1 = M \leq N$.
-

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **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 Iter.
- **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.
- **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 first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted 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.
- **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*.

Returns 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).

```
template<typename Iter, typename Sent, typename Iter2, typename T, typename Reduce, typename Convert>
T transform_reduce(Iter first, Sent last, Iter2 first2, T init, Reduce &&red_op, Convert &&conv_op)
    Returns GENERALIZED_SUM(red_op, init, conv_op(*first), ..., conv_op(*first + (last - first) - 1)).
```

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.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

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, ..., *b*N may be any permutation of *a*1, ..., *a*N and
 - 1 < K+1 = M <= N.

Template Parameters

- **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 Iter.
- **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.
- **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

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted 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.
- **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 & <i>a</i> , const Type2 & <i>b</i>);

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`.

Returns The `transform_reduce` algorithm returns `T`. 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).

```
template<typename ExPolicy, typename Rng, typename T, typename Reduce, typename Convert>
hpx::parallel::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.

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.

Note: Complexity: $O(last - first)$ applications of the predicates `red_op` and `conv_op`.

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 Parameters

- **ExPolicy** – The type of the execution policy to use (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).
- **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*.

Returns 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).

```
template<typename Rng, typename T, typename Reduce, typename Convert>
T transform_reduce(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 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.

Note: Complexity: O(*last - first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(*op*, *a1*, ..., *aN*) is defined as follows:

- *a1* when *N* is 1

- $\text{op}(\text{GENERALIZED_SUM}(\text{op}, \text{b}_1, \dots, \text{b}_K), \text{GENERALIZED_SUM}(\text{op}, \text{b}_M, \dots, \text{b}_N))$, where:
 - $\text{b}_1, \dots, \text{b}_N$ may be any permutation of $\text{a}_1, \dots, \text{a}_N$ and
 - $1 < K+1 = M \leq N$.
-

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **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

- **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*.

Returns The *transform_reduce* algorithm returns *T*. 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).

```
template<typename ExPolicy, typename Rng, typename Iter2, typename T>
hpx::parallel::util::detail::algorithm_result<ExPolicy, T>::type transform_reduce(ExPolicy &&policy, Rng &&rng, Iter2 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.

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.

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.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter2** – 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.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.

Returns 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 **Rng**, typename **Iter2**, typename **T**>
T transform_reduce(*Rng* &&*rng*, *Iter2* *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*.

Note: Complexity: $O(last - first)$ applications of the predicate *op2*.

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.
- **Iter2** – 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

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.

Returns The *transform_reduce* algorithm returns *T*.

```
template<typename ExPolicy, typename Rng, typename Iter2, typename T, typename Reduce, typename Convert>
hpx::parallel::util::detail::algorithm_result<ExPolicy, T>::type transform_reduce(ExPolicy &&policy, Rng &&rng, Iter2 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.

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.

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.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter2** – 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.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.
- **red_op** – Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *op2*. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Ret` must be such that it can be implicitly converted to a type of *T*.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to an object for the second argument type of *op1*.

Returns 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 Rng, typename Iter2, typename T, typename Reduce, typename Convert>
T transform_reduce(Rng &&rng, Iter2 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.

Note: Complexity: O(*last - first*) applications of the predicate *op2*.

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.
- **Iter2** – 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

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.
- **red_op** – Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *op2*. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Ret` must be such that it can be implicitly converted to an object for the second argument type of `op1`.

Returns The `transform_reduce` algorithm returns `T`.

hpx/parallel/container_algorithms/uninitialized_copy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

Returns 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`

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.

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.

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.

Returns 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_traits<Rng2>::iterator_type>`

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

Returns 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>>`

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.

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.

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

Returns 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(InIter first1, Size count,
                                                               FwdIter first2, Sent2 last2)
```

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object 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.

Returns 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`

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.

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.

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.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns 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.

hpx/parallel/container_algorithms/uninitialized_default_construct.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 a 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.

Returns 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 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.

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.

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.
- **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.

Returns 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.

Returns 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 uninitialized

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.

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.

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

Returns 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 a 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.

Returns 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.

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.

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.

Parameters

- **policy** – The execution policy 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.

Returns 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.

hpx/parallel/container_algorithms/uninitialized_fill.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 a 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.

Returns 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, typename T>
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.

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.

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 a 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.

Returns 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.

Returns 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(**ExPolicy** &&**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.

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.

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.

Returns 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_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.

Returns 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.

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.

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.

Returns 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.

hpx/parallel/container_algorithms/uninitialized_move.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

Returns 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.

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.

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.

Returns 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_trait
```

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

Returns 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.

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.

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

Returns 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 Sent2InIter, 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.

Returns 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.

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.

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.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns 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.

[hpx/parallel/container_algorithms/uninitialized_value_construct.hpp](#)

See *Public API* for a list of names and headers that are part of the public HPX API.

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 a 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.

Returns The *uninitialized_value_construct* algorithm returns a returns *FwdIter*. The *uninitialized_value_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

```
template<typename ExPolicy, typename FwdIter, typename Sent>
parallel::util::detail::algorithm_result<ExPolicy, 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.

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.

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.
- **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.

Returns 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.

Returns 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 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(ExPolicy &&policy, Rng &&rng)
```

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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 consumed

Returns 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 a 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.

Returns 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.

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.

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.

Parameters

- **policy** – The execution policy 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.

Returns 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.

hpx/parallel/container_algorithms/unique.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **ranges**

Functions

```
template<typename FwdIter, typename Sent, typename Pred = ranges::equal_to, typename Proj = parallel::util::projection_identity>
subrange_t<FwdIter, Sent> unique(FwdIter first, Sent last, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked 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 a 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 a binary predicate which returns `true` for the required elements. The signature of this predicate should be equivalent to:

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

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The types `Type1` and `Type2` must be such that objects of types `FwdIter` can be dereferenced and then implicitly converted to both `Type1` and `Type2`

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns 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 = ranges::equal_to,
        typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter, Sent>>::type unique(ExPolicy
    &&policy,
    FwdIter
    first, Sent
    last, Pred
    &&pred =
    Pred(),
    Proj
    &&proj =
    Proj())
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel `unique` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

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.

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 a 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.

Returns 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 = ranges::equal_to, typename Proj = parallel::util::projection_identity>
subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> unique(Rng
    &&rng,
    Pred
    &&pred =
    Pred(),
    Proj
    &&proj =
    Proj())
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range *rng* and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked 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 a 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 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.

Returns The *unique* algorithm returns *subrange_t<typename hpx::traits::range_iterator_t<Rng>, 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.

```
template<typename ExPolicy, typename Rng, typename Pred = ranges::equal_to, typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> unique(hpx::traits::range_iterator_t<Rng> first, hpx::traits::range_iterator_t<Rng> last, 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 with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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 $N = \text{std}::\text{distance}(\text{begin}(\text{rng}), \text{end}(\text{rng}))$.

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 *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to `std::equal_to<>`
- **Proj** – The type of an optional projection function. This defaults to `util::projection_identity`

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique* 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<typename hpx::traits::range_iterator<Rng>::type, hpx::traits::range_iterator_t<Rng>>` otherwise. The *unique* algorithm returns an object {ret, last}, where *ret* is a past-the-end iterator for a new subrange.

```
template<typename InIter, typename Sent, typename O, typename Pred = ranges::equal_to, typename Proj = parallel::util::projection_identity>
unique_copy_result<InIter, O> unique_copy(InIter first, Sent last, O dest, Pred &&pred = Pred(), Proj &&proj = Proj())
```

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked 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.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **O** – The type of the iterator representing the destination range (deduced).
- **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 sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **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 *InIter1* 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.

Returns The *unique_copy* algorithm returns a returns unique_copy_result<InIter, O>. The *unique_copy* algorithm returns an in_out_result with the source iterator to one past the last element and out containing the destination iterator to the end of the *dest* range.

```
template<typename ExPolicy, typename FwdIter, typename Sent, typename O, typename Pred = ranges::equal_to, typename Proj = parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, unique_copy_result<FwdIter, O>>::type unique_copy(ExPolicy
    &&policy,
    FwdIter
    first,
    Sent
    last,
    O
    dest,
    Pred
    &&pred
    =
    Pred(),
    Proj
    &&proj
    =
    Proj())
```

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

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 a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter1.
- **O** – The type of the iterator representing the destination range (deduced).
- **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 sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *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.

Returns The *unique_copy* algorithm returns areturns hpx::future< *unique_copy_result*<*FwdIter*, *O*>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *unique_copy_result*<*FwdIter*, *O*> otherwise. The *unique_copy* algorithm returns an *in_out_result* with the source iterator to one past the last element and out containing the destination iterator to the end of the *dest* range.

```
template<typename Rng, typename O, typename Pred = ranges::equal_to, typename Proj = parallel::util::projection_identity>
```

```
unique_copy_result<hpx::traits::range_iterator_t<Rng>, O> unique_copy(Rng &&rng, O dest, Pred
&&pred = Pred(), Proj
&&proj = Proj())
```

Copies the elements from the range *rng*, to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **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 a 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

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have *const&*, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique_copy* algorithm returns *unique_copy_result<hpx::traits::range_iterator_t<Rng>, O>*. The *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

```
template<typename ExPolicy, typename Rng, typename O, typename Pred = ranges::equal_to,
typename Proj = parallel::util::projection_identity>
```

`parallel::util::detail::algorithm_result<ExPolicy, unique_copy_result<hpx::traits::range_iterator_t<Rng>, O>>::type unique`

Copies the elements from the range *rng*, to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.

Note: Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – 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 *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *util::projection_identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have `const&`, but the function must not modify the objects passed to it. The type `Type` must be such that an object of type `FwdIter1` can be dereferenced and then implicitly converted to `Type`.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate is invoked.

Returns The `unique_copy` algorithm returns a `hpx::future<unique_copy_result< hpx::traits::range_iterator_t<Rng>, O>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `unique_copy_result< hpx::traits::range_iterator_t<Rng>, O>` 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.

hpx/parallel/util/low_level.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **util**

Functions

template<typename **Value**, typename ...**Args**>
inline 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

- **ptr** – [in] : pointer to the memory where to create the object
- **args** – [in] : arguments to the constructor

template<typename **Value**>

inline void **destroy_object**(**Value** *ptr)

destroy an object in the memory specified by ptr

Template Parameters **Value** – : typename of the object to create

Parameters **ptr** – [in] : pointer to the object to destroy

template<typename **Iter**, typename **Sent**>

inline 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

Parameters

- **r** – [in] : range of elements not initialized
- **val** – [in] : object used for the initialization

Returns range initialized

template<typename **Value**, typename ...**Args**>

inline 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

- **ptr** – [in] : pointer to the memory where to create the object
- **args** – [in] : arguments to the constructor

```
template<typename Iter1, typename Sent1, typename Iter2>
inline 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

- **itdest** – [in] : iterator to the final place of the objects
- **R** – [in] : range to move

```
template<typename Iter, typename Sent, typename Value = typename
std::iterator_traits<Iter>::value_type>
inline 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

- **ptr** – [in] : pointer to the memory where to create the object
- **R** – [in] : range to move

```
template<typename Iter, typename Sent>
inline 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

- **ptr** – [in] : pointer to the memory where to construct the object
- **R** – [in] : range to move

```
template<typename Iter1, typename Sent1, typename Iter2, typename Compare>
inline 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

- **buf1** – [in] : iterator to the first element in the first buffer
- **end_buf1** – [in] : final iterator of first buffer
- **buf2** – [in] : iterator to the first iterator to the second buffer
- **end_buf2** – [in] : final iterator of the second buffer
- **buf_out** – [in] : buffer where move the elements merged
- **comp** – [in] : comparison object

```
template<typename Iter, typename Sent, typename Value, typename Compare>
inline 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

- **first1 – [in]** : iterator to the first element in the first buffer
- **last – [in]** : last iterator of the first buffer
- **first2 – [in]** : iterator to the first element to the second buffer
- **last2 – [in]** : final iterator of the second buffer
- **it_out – [in]** : uninitialized buffer where move the elements merged
- **comp – [in]** : comparison object

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Compare**>
 inline **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

- **buf1 – [in]** : iterator to the first element of the first buffer
- **end_buf1 – [in]** : iterator to the last element of the first buffer
- **buf2 – [in]** : iterator to the first element of the second buffer
- **end_buf2 – [in]** : iterator to the last element of the second buffer
- **buf_out – [in]** : iterator to the first element to the buffer where put the result
- **comp – [in]** : 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

- **src1 – [in]** : iterator to the first element of the first buffer
- **end_src1 – [in]** : last iterator of the first buffer
- **src2 – [in]** : iterator to the first element of the second buffer
- **end_src2 – [in]** : last iterator of the second buffer
- **aux – [in]** : iterator to the first element of the auxiliary buffer
- **comp – [in]** : object for to Compare elements

Throws

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Compare**>
 inline 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

- **src1 – [in]** iterator to the first position of the first buffer
- **src2 – [in]** final iterator of the first buffer and first iterator of the second buffer
- **end_src2 – [in]** : final iterator of the second buffer
- **buf – [in]** : iterator to buffer used as auxiliary memory
- **comp – [in]** : object for to Compare elements

Throws

hpx/parallel/util/merge_four.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **util**

Functions

template<typename **Iter**, typename **Sent**, typename **Compare**>

inline 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.

Parameters

- **it1** – [in] : iterator to the first element
- **pos1** – [in] : position of the object pointed by it1
- **it2** – [in] : iterator to the second element
- **pos2** – [in] : position of the element pointed by it2
- **comp** – [in] : comparison object

Returns result of the comparison

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.

Parameters

- **dest** – [in] 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
- **R** – [in] : array of ranges to merge
- **nrange_input** – [in] : number of ranges in R
- **comp** – [in] : comparison object

Returns range with all the elements move with the size adjusted

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.

Parameters

- **dest** – [in] 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
- **R** – [in] : array of ranges to merge
- **nrange_input** – [in] : number of ranges in vrangle_input
- **comp** – [in] : comparison object

Returns range with all the elements move with the size adjusted

hpx/parallel/util/merge_vector.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

Parameters

- **dest** – [in] : range where move the elements merged
- **v_input** – [in] : vector of ranges to merge
- **v_output** – [in] : vector of ranges obtained
- **comp** – [in] : comparison object

Returns range with all the elements moved

```
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

Parameters

- **dest** – [in] : range where move the elements merged
- **v_input** – [in] : vector of ranges to merge
- **v_output** – [in] : vector of ranges obtained
- **comp** – [in] : comparison object

Returns range with all the elements moved

```
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 **range_input** – [in] : range including all the ranges of v_input
Param

hpx/parallel/util/nbits.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **parallel**

namespace **util**

Functions

```
inline constexpr std::uint32_t nbits32(std::uint32_t num) noexcept
```

Obtain the number of bits equal or greater than num.

Parameters num -

Throws none –

Returns Number of bits

ne constexpr `std::uint32_t` **nbits64**(`std::uint64_t` num)

obtain the number

Parameters num – [in] :

[hpx/parallel/util/projection_identity.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API

namespace **hpx**

TypeDefs

```
using identity = hpx::parallel::util::projection_identity

namespace parallel

namespace util

    struct projection_identity
        #include <projection_identity.hpp> this represents the projection identity
```

Public Types

```
using is_transparent = std::true_type
```

Public Functions

```
template<typename T>
inline constexpr T &&operator() (T &&val) const noexcept
```

hpx/parallel/util/range.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

```
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
Parameters

- it1 – [in] : first range
- it2 – [in] : second range

Returns range resulting of the concatenation

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2>
inline range<Iter2, Iter2> init_move(range<Iter2, Sent2> const &dest, range<Iter1, Sent1> const
&src)
    Move objects from the range src to dest.
Parameters

- dest – [in] : range where move the objects
- src – [in] : range from where move the objects

Returns range with the objects moved and the size adjusted

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2>
inline 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.
Parameters

- dest – [in] : range where move and create the objects
- src – [in] : range from where move the objects

Returns range with the objects moved and the size adjusted

template<typename Iter, typename Sent>
inline void destroy_range(range<Iter, Sent> r)
    destroy a range of objects
Parameters r – [in] : range to destroy

template<typename Iter, typename Sent>
inline 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
Parameters

- r – [in] : range of elements not initialized
- val – [in] : object used for the initialization

Returns range initialized

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename
Compare>
inline bool is_mergeable(range<Iter1, Sent1> const &src1, range<Iter2, Sent2> const &src2,
Compare comp)
    : indicate if two ranges have a possible merge
Parameters

- src1 – [in] : first range
- src2 – [in] : second range
- comp – [in] : object for to compare elements

Throws

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3,
typename Sent3, typename Compare>
```

```
inline 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.

Parameters

- **dest** – [in] : 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
- **src1** – [in] : first range to merge
- **src2** – [in] : second range to merge
- **comp** – [in] : comparison object

Returns range with the elements merged and the size adjusted

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Value, typename Compare>
inline 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.

Parameters

- **dest** – [in] : 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
- **src1** – [in] : first range to merge
- **src2** – [in] : second range to merge
- **comp** – [in] : comparison object

Returns range with the elements merged and the size adjusted

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Value, typename Compare>
inline 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

Parameters

- **dest** – [in] : range where finish the two buffers merged
- **src1** – [in] : first range to merge in a separate memory
- **src2** – [in] : second range to merge, in the final part of the range where deposit the final results
- **comp** – [in] : object for compare two elements of the type pointed by the Iter1 and Iter2

Returns : range with the two buffers merged

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Sent3, typename Compare>
bool in_place_merge_uncontiguous(range<Iter1, Sent1> const &src1, range<Iter2, Sent2> const &src2, range<Iter3, Sent3> &aux, Compare comp)
```

: merge two non contiguous buffers src1 , src2, using the range aux as auxiliary memory

Parameters

- **src1** – [in] : first range to merge
- **src2** – [in] : second range to merge
- **aux** – [in] : auxiliary range used in the merge
- **comp** – [in] : object for to compare elements

Throws

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Value, typename Compare>
```

```
inline 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

- **src1** – [in] : first range to merge
- **src2** – [in] : second range to merge
- **buf** – [in] : auxiliary memory used in the merge
- **comp** – [in] : object for to compare elements

Throws

```
template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
```

```
inline 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

- **first** – [in] : iterator to the first element
- **last** – [in] : iterator to the element after the last in the range
- **comp** – [in] : object for to compare elements

Throws

asio

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/asio/asio_util.hpp

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, bool
force_ipv4 = false)

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, bool force_ipv4 = false)

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.

inline 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.

inline endpoint_iterator_type accept_end()

bool split_ip_address(std::string const &v, std::string &host, std::uint16_t &port)

```

assertion

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/assertion/evaluate_assert.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **assertion**

hpx/assertion/source_location.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

Defines

`HPX_CURRENT_SOURCE_LOCATION()`

namespace `hpx`

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. The `source_location` class represents certain information about the source code, such as file names, line numbers, and function names. Previously, functions that desire to obtain this information about the call site (for logging, testing, or debugging purposes) must use macros so that predefined macros like `__FILE__` and `__LINE__` are expanded in the context of the caller. The `source_location` class provides a better alternative. `source_location` meets the `DefaultConstructible`, `CopyConstructible`, `CopyAssignable` and `Destructible` requirements. Lvalue of `source_location` meets the `Swappable` requirement. Additionally, the following conditions are true:

•

`std::is_nothrow_move_constructible_v<std::source_location>`

•

`std::is_nothrow_move_assignable_v<std::source_location>`

•

`std::is_nothrow_swappable_v<std::source_location>`

It is intended that `source_location` has a small size and can be copied efficiently. It is unspecified whether the copy/move constructors and the copy/move assignment operators of `source_location` are trivial and/or `constexpr`.

Public Functions

inline `constexpr std::uint_least32_t line()` const noexcept

return the line number represented by this object

inline `constexpr std::uint_least32_t column()` const noexcept

return the column number represented by this object

inline `constexpr const char *file_name()` const noexcept

return the file name represented by this object

```
inline constexpr const char *function_name() const noexcept
    return the name of the function represented by this object, if any
```

Public Members

```
const char *filename
```

```
std::uint_least32_t line_number
```

```
const char *functionname
```

```
namespace assertion
```

Typedefs

```
using instead = hpx::source_location
```

hpx/modules/assertion.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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()`.

Asserts are enabled if `HPX_DEBUG` is set. This is the default for `CMAKE_BUILD_TYPE=Debug`

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`

HPX_ASSERT_MSG(expr, msg)

See also:

`HPX_ASSERT`

```
namespace hpx
```

```
namespace assertion
```

Typedefs

```
using assertion_handler = void (*)(hpx::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.

hpx/async_base/async.hpp

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>
decltype(auto) async(F &&f, Ts&&... ts)
```

The function template *async* runs the function *f* asynchronously (potentially in a separate thread which might be a part of a thread pool) and returns an *hpx::future* that will eventually hold the result of that function call. If no policy is defined, *async* behaves as if it is called with policy being *hpx::launch::async* | *hpx::launch::deferred*. Otherwise, it calls a function *f* with arguments *ts* according to a specific launch policy.

- If the *async* flag is set (i.e. `(policy & hpx::launch::async) != 0`), then *async* executes the callable object *f* on a new thread of execution (with all thread-locals initialized) as if spawned by *hpx::thread(std::forward<F>(f), std::forward<Ts>(ts)...)*, except that if the function *f* returns a value or throws an exception, it is stored in the shared state accessible through the *hpx::future* that *async* returns to the caller.
- If the *deferred* flag is set (i.e. `(policy & hpx::launch::deferred) != 0`), then *async* converts *f* and *ts...* the same way as by *hpx::thread* constructor, but does not spawn a new thread of execution. Instead, lazy evaluation is performed: the first call to a non-timed wait function on the *hpx::future* that *async* returned to the caller will cause the copy of *f* to be invoked (as an rvalue) with the copies of *ts...* (also passed as rvalues) in the current thread (which does not have to be the thread that originally called *hpx::async*). The result or exception is placed in the shared state associated with the future and only then it is made ready. All further accesses to the same *hpx::future* will return the result immediately.

- If neither `hpx::launch::async` nor `hpx::launch::deferred`, nor any implementation-defined policy flag is set in policy, the behavior is undefined. If more than one flag is set, it is implementation-defined which policy is selected. For the default (both the `hpx::launch::async` and `hpx::launch::deferred` flags are set in policy), standard recommends (but doesn't require) utilizing available concurrency, and deferring any additional tasks. In any case, the call to `hpx::async` synchronizes-with (as defined in `std::memory_order`) the call to `f`, and the completion of `f` is sequenced-before making the shared state ready. If the `async` policy is chosen, the associated thread completion synchronizes-with the successful return from the first function that is waiting on the shared state, or with the return of the last function that releases the shared state, whichever comes first. If `std::decay<Function>::type` or each type in `std::decay<Ts>::type` is not constructible from its corresponding argument, the program is ill-formed.

Parameters

- `f` – Callable object to call
- `ts...` – parameters to pass to `f`

Returns `hpx::future` referring to the shared state created by this call to `hpx::async`.

hpx/async_base/dataflow.hpp

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>
auto dataflow(F &&f, Ts&&... ts)
```

The function template `dataflow` runs the function `f` asynchronously (potentially in a separate thread which might be a part of a thread pool) and returns a `hpx::future` that will eventually hold the result of that function call. Its behavior is similar to `hpx::async` with the exception that if one of the arguments is a `future`, then `hpx::dataflow` will wait for the `future` to be ready to launch the thread. Hence, the operation is delayed until all the arguments are ready.

hpx/async_base/launch_policy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

```
struct launch : public detail::policy_holder<>
```

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

Public Functions

```
inline constexpr launch() noexcept
    Default constructor. This creates a launch policy representing all possible launch modes

inline constexpr launch(detail::async_policy p) noexcept
    Create a launch policy representing asynchronous execution.

inline constexpr launch(detail::fork_policy p) noexcept
    Create a launch policy representing asynchronous execution. The new thread is executed in a preferred
    way

inline constexpr launch(detail::sync_policy p) noexcept
    Create a launch policy representing synchronous execution.

inline constexpr launch(detail::deferred_policy p) noexcept
    Create a launch policy representing deferred execution.

inline constexpr launch(detail::apply_policy p) noexcept
    Create a launch policy representing fire and forget execution.

template<typename F>
inline constexpr launch(detail::select_policy<F> const &p) noexcept
    Create a launch policy representing fire and forget execution.

template<typename Launch, typename Enable =
std::enable_if_t<hpz::traits::is_launch_policy_v<Launch>>>
inline constexpr launch(Launch l, threads::thread_priority priority, threads::thread_stacksize stacksize,
                      threads::thread_schedule_hint hint) noexcept
```

Public Static Attributes

```
static const detail::async_policy async
    Predefined launch policy representing asynchronous execution.

static const detail::fork_policy fork
    Predefined launch policy representing asynchronous execution. The new thread is executed in a pre-
    ferred way

static const detail::sync_policy sync
    Predefined launch policy representing synchronous execution.

static const detail::deferred_policy deferred
    Predefined launch policy representing deferred execution.

static const detail::apply_policy apply
    Predefined launch policy representing fire and forget execution.

static const detail::select_policy_generator select
    Predefined launch policy representing delayed policy selection.
```

Friends

```
inline friend launch tag_invoke(hpx::execution::experimental::with_priority_t, launch const &policy,  
                                threads::thread_priority priority) noexcept  
  
inline friend constexpr friend hpx::threads::thread_priority tag_invoke (hpx::execution::experimental::  
                                launch const &policy) noexcept  
  
inline friend launch tag_invoke(hpx::execution::experimental::with_stacksize_t, launch const &policy,  
                                threads::thread_stacksize stacksize) noexcept  
  
inline friend constexpr friend hpx::threads::thread_stacksize tag_invoke (hpx::execution::experimental::  
                                launch const &policy) noexcept  
  
inline friend launch tag_invoke(hpx::execution::experimental::with_hint_t, launch const &policy,  
                                threads::thread_schedule_hint hint) noexcept  
  
inline friend constexpr friend hpx::threads::thread_schedule_hint tag_invoke (hpx::execution::experimental::  
                                launch const &policy) noexcept
```

hpx/async_base/sync.hpp

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>  
auto sync(F &&f, Ts&&... ts) ->  
    decltype(detail::sync_dispatch<std::decay_t<F>>::call(HPX_FORWARD(F, f),  
              HPX_FORWARD(Ts, ts...))
```

The function template *sync* runs the function *f* synchronously and returns an *hpx::future* that will eventually hold the result of that function call.

async_combinators

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/async_combinators/split_future.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Top level HPX namespace.

Functions

```
template<typename ...Ts>
inline 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*.

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>*

Returns 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.

```
template<typename T>
inline 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*.

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

Returns 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.

hpx/async_combinators/wait_all.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Top level HPX namespace.

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(hpx::future<T> const &f)
```

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 the future has become ready. The input future is still valid after *wait_all* returns.

Note: The function *wait_all* will rethrow any exceptions captured by the future while becoming ready. If this behavior is undesirable, use *wait_all_nothrow* instead.

Parameters **f** – A *future* or *shared_future* 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.

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*.

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

hpx/async_combinators/wait_any.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Top level HPX namespace.

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`.

hpx/async_combinators/wait_each.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace hpx

Top level HPX namespace.

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*.

hpx/async_combinators/wait_some.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Top level HPX namespace.

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*.

`hpx/async_combinators/when_all.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace `hpx`

Top level HPX namespace.

Functions

```
template<typename InputIter, typename Container = vector<future<typename
std::iterator_traits<InputIter>::value_type>>>
hpx::future<Container> when_all(InputIter first, InputIter last)

function when_all creates a future object that becomes ready when all elements in a set of future and
shared_future objects become ready. It is an operator allowing to join on the result of all given futures. It
AND-composes all given future objects and returns a new future object representing the same list of futures
after they finished executing.
```

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.

Returns 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.

`template<typename Range>`

hpx::future<Range> **when_all**(*Range* &&*values*)

function *when_all* creates a future object that becomes ready when all elements in a set of *future* and *shared_future* objects become ready. It is an operator allowing to join on the result of all given futures. It AND-composes all given future objects and returns a new future object representing the same list of futures after they finished executing.

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.

Returns 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.

template<typename ...T>

hpx::future<hpx::tuple<hpx::future<T>...>> **when_all**(*T* &&... *futures*)

function *when_all* creates a future object that becomes ready when all elements in a set of *future* and *shared_future* objects become ready. It is an operator allowing to join on the result of all given futures. It AND-composes all given future objects and returns a new future object representing the same list of futures after they finished executing.

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.

Returns 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.

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)

function *when_all* creates a future object that becomes ready when all elements in a set of *future* and *shared_future* objects become ready. It is an operator allowing to join on the result of all given futures. It AND-composes all given future objects and returns a new future object representing the same list of futures after they finished executing.

Note: As long as *ec* is not pre-initialized to *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*.

Throws 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.

Returns 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.

hpx/async_combinators/when_any.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

Top level HPX namespace.

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)

function when_any creates a future object that becomes when at least one element in a set of future and shared_future objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.
```

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.

Returns 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.

template<typename Range>

`future<when_any_result<Range>> when_any(Range &values)`

function `when_any` creates a future object that becomes when at least one element in a set of `future` and `shared_future` objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.

Parameters `values` – [in] A range holding an arbitrary amount of `futures` or `shared_future` objects for which `when_any` should wait.

Returns 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.

template<typename ...T>

`future<when_any_result<tuple<future<T>...>>> when_any(T&&... futures)`

function `when_any` creates a future object that becomes when at least one element in a set of `future` and `shared_future` objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.

Parameters `futures` – [in] An arbitrary number of `future` or `shared_future` objects, possibly holding different types for which `when_any` should wait.

Returns 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.

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)`

function `when_any_n` creates a future object that becomes when at least one element in a set of `future` and `shared_future` objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.

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`.

Returns 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.

```
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`

hpx/async_combinators/when_each.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace `hpx`

Top level HPX namespace.

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.

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.

Returns Returns a future representing the event of all input futures being ready.

```
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.

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.

Returns Returns a future representing the event of all input futures being ready.

```
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.

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.

Returns Returns a future representing the event of all input futures being ready.

```
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.

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*.

Returns Returns a future holding the iterator pointing to the first element after the last one.

hpx/async_combinators/when_some.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace hpx

Top level HPX namespace.

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.

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.

Returns 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.

```
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.

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.

Returns 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.

```
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.

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.

Returns 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.

```
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.

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*.

Returns 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.

```
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*.

hpx/async_cuda/cublas_executor.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/async_cuda/cuda_executor.hpp

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_executor** : public `hpx::cuda::experimental::cuda_executor_base`

Public Functions

inline explicit **cuda_executor**(`std::size_t` device, `bool` event_mode = true)

inline ~**cuda_executor**()

template<typename **F**, typename ...**Ts**>

inline decltype(auto) friend **tag_invoke**(`hpx::parallel::execution::post_t`, `cuda_executor` const &exec, **F** &&f, **Ts**&&... ts)

template<typename **F**, typename ...**Ts**>

inline decltype(auto) friend **tag_invoke**(`hpx::parallel::execution::async_execute_t`, `cuda_executor` const &exec, **F** &&f, **Ts**&&... ts)

Protected Functions

```
template<typename R, typename ...Params, typename ...Args>
inline void apply(R (*cuda_function)(Params...), Args&&... args) const

template<typename R, typename ...Params, typename ...Args>
inline hpx::future<void> async(R (*cuda_kernel)(Params...), Args&&... args) const

struct cuda_executor_base
    Subclassed by hpx::cuda::experimental::cuda_executor
```

Public Types

```
using future_type = hpx::future<void>
```

Public Functions

```
inline cuda_executor_base(std::size_t device, bool event_mode)
inline future_type get_future() const
```

Protected Attributes

```
int device_

bool event_mode_

cudaStream_t stream_

std::shared_ptr<hpx::cuda::experimental::target> target_

namespace parallel

namespace execution

async_mpi
```

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/async_mpi/mpi_executor.hpp

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 **executor**

Public Types

 using **execution_category** = *hpx::execution::parallel_execution_tag*

 using **executor_parameters_type** = *hpx::execution::static_chunk_size*

Public Functions

 inline explicit constexpr **executor**(MPI_Comm communicator = MPI_COMM_WORLD)

 template<typename **F**, typename ...**Ts**>

 inline decltype(auto) friend **tag_invoke**(*hpx::parallel::execution::async_execute_t*, **executor** const &exec, **F** &&f, **Ts**&&... ts)

 inline *std::size_t* **in_flight_estimate**() const

Private Members

 MPI_Comm **communicator_**

 namespace **parallel**

 namespace **execution**

hpx/async_mpi/transform_mpi.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **mpi**

 namespace **experimental**

Variables

`hpx::mpi::experimental::transform_mpi_t transform_mpi`

```
struct transform_mpi_t : public hpx::functional::detail::tag_fallback<transform_mpi_t>
```

Friends

```
template<typename Sender,
typename F> inline friend constexpr friend auto tagFallback_invoke (transform_mpi_t,
Sender &&s, F &&f)

template<typename F> inline friend constexpr friend auto tagFallback_invoke (transform_mpi_t,
F &&f)
```

cache

See [Public API](#) for a list of names and headers that are part of the public HPX API.

`hpx/cache/local_cache.hpp`

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 = policies::always<Entry>, typename CacheStorage = std::map<Key, Entry>,
typename Statistics = statistics::no_statisticslocal_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

using **key_type** = `Key`

using **entry_type** = `Entry`

using **update_policy_type** = `UpdatePolicy`

using **insert_policy_type** = `InsertPolicy`

using **storage_type** = `CacheStorage`

using **statistics_type** = `Statistics`

using **value_type** = typename `entry_type`::value_type

using **size_type** = typename `storage_type`::size_type

using **storage_value_type** = typename `storage_type`::value_type

Public Functions

```
inline 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) = default

inline constexpr *size_type* **size**() const noexcept

Return current size of the cache.

Returns The current size of this cache instance.

inline constexpr *size_type* **capacity**() const noexcept

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*.

Returns 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.

inline bool **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.

Returns 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.

inline 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.

Parameters **k** – [in] The key for the entry which should be looked up in the cache.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline 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.

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.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline 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.

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.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline 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.

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.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline 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.

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.

Returns This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

inline bool **insert**(*key_type* const &*k*, *value_type* &&*val*)

template<typename **Entry_**, std::enable_if_t<std::is_convertible_v<std::decay_t<**Entry_entry_type>, int> = 0>**

inline bool **insert**(*key_type* const &*k*, *Entry_* &&*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.

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.

Returns This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

```
template<typename Value, std::enable_if_t<std::is_convertible_v<std::decay_t<Value>,  
        value_typeinline bool update(key_type const &k, Value &&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.

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.

Returns 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.

```
template<typename F, typename Value, typename =  
        std::enable_if_t<std::is_convertible_v<std::decay_t<Value>, value_type>>>  
inline bool update_if(key_type const &k, Value &&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.

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.

Returns 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.

```
template<typename Entry, std::enable_if_t<std::is_convertible_v<std::decay_t<Entry>,  
        entry_type>, int> = 0>
```

```
inline bool update(key_type const &k, Entry_ &&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.

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.

Returns 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.

```
template<typename Func = policies::always<storage_value_type>>
inline size_type erase(Func &&ep = Func())
```

Remove stored entries from the cache for which the supplied function object returns true.

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.

Returns 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).

```
inline 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.

Returns 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).

```
inline void clear()
```

Clear the cache.

Unconditionally removes all stored entries from the cache.

```
inline constexpr statistics_type const &get_statistics() const noexcept
```

Allow to access the embedded statistics instance.

Returns This function returns a reference to the statistics instance embedded inside this cache

```
inline statistics_type &get_statistics() noexcept
```

Protected Functions

```
inline bool free_space(long num_free)
```

Private Types

```
using iterator = typename storage_type::iterator
```

```
using const_iterator = typename storage_type::const_iterator
```

```
using heap_type = std::deque<iterator>
```

```
using heap_iterator = typename heap_type::iterator
```

```
using adapted_update_policy_type = adapt<UpdatePolicy, iterator>
```

```
using update_on_exit = typename statistics_type::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

```
inline explicit adapt(Func const &f)  
inline explicit adapt(Func &&f) noexcept  
inline bool operator()(Iterator const &lhs, Iterator const &rhs) const
```

Public Members

Func **f_**

hpx/cache/lru_cache.hpp

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 Statistics = statistics::no_statistics>  
class lru_cache  
#include <hpx/cache/lru_cache.hpp> The lru_cache implements the basic functionality needed  
for a local (non-distributed) LRU cache.
```

Template Parameters

- **Key** – The type of the keys to use to identify the entries stored in the cache
- **Entry** – The type of the items to be held in the cache.
- **Statistics** – A (optional) type allowing to collect some basic statistics about the operation of the cache instance. The type must conform to the CacheStatistics concept. The default value is the type *statistics::no_statistics* which does not collect any numbers, but provides empty stubs allowing the code to compile.

Public Types

using **key_type** = *Key*

using **entry_type** = *Entry*

using **statistics_type** = *Statistics*

using **entry_pair** = *std::pair<key_type, entry_type>*

using **storage_type** = *std::list<entry_pair>*

```
using map_type = std::map<Key, typename storage_type::iterator>
```

```
using size_type = std::size_t
```

Public Functions

inline **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) = default

inline constexpr size_type **size()** const noexcept

Return current size of the cache.

Returns The current size of this cache instance.

inline constexpr size_type **capacity()** const noexcept

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*.

Returns 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.

inline 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.

inline bool **holds_key(key_type const &key)** 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.

Parameters **k** – [in] The key for the entry which should be looked up in the cache.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline 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.

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.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

```
inline bool get_entry(key_type const &key, entry_type const &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.

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.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

```
template<typename Entry_, typename =  
std::enable_if_t<std::is_convertible_v<std::decay_t<Entry_>, entry_type>>>  
inline bool insert(key_type const &key, Entry_ &&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.

```
template<typename Entry_, typename =  
std::enable_if_t<std::is_convertible_v<std::decay_t<Entry_>, entry_type>>>  
inline void update(key_type const &key, Entry_ &&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, typename Entry_,  
std::enable_if_t<std::is_convertible_v<std::decay_t<Entry_>, entry_type>, int> = 0>
```

```
inline bool update_if(key_type const &key, Entry_ &&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.

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.

Returns 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.

```
template<typename Func>
inline size_type erase(Func const &ep)
```

Remove stored entries from the cache for which the supplied function object returns true.

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*.

Returns 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).

```
inline size_type erase()
```

Remove all stored entries from the cache.

Returns 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).

```
inline size_type clear()
```

Clear the cache.

Unconditionally removes all stored entries from the cache.

```
inline constexpr statistics_type const &get_statistics() const noexcept
```

Allow to access the embedded statistics instance.

Returns This function returns a reference to the statistics instance embedded inside this cache

```
inline statistics_type &get_statistics() noexcept
```

Private Types

```
using update_on_exit = typename statistics_type::update_on_exit
```

Private Functions

```
template<typename Entry_, typename =
std::enable_if_t<std::is_convertible_v<std::decay_t<Entry_>, entry_type>>>
inline void insert_nonexist(key_type const &key, Entry_ &&entry)

inline void touch(typename storage_type::iterator it)

inline void evict()
```

Private Members

```
size_type max_size_
```

```
size_type current_size_ = 0
```

```
storage_type storage_
```

```
map_type map_
```

```
statistics_type statistics_
```

hpx/cache/entries/entry.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

using **value_type** = Value

Public Functions

entry() = default

Any cache entry has to be default constructible.

inline explicit **entry**(*value_type* const &val)
noexcept(*std*::is_nothrow_copy_constructible_v<*value_type*>)

Construct a new instance of a cache entry holding the given value.

inline explicit **entry**(*value_type* &&val) noexcept

Construct a new instance of a cache entry holding the given value.

inline constexpr bool **touch()** const noexcept

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

Returns 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

inline constexpr bool **insert()** const noexcept

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

Returns This function should return *true* if the entry should be added to the cache, otherwise it should return *false*.

inline constexpr bool **remove()** const noexcept

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

Returns 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.

```
inline constexpr std::size_t get_size() const noexcept
```

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.

```
inline value_type &get() noexcept
```

Get a reference to the stored data value.

Note: This function is part of the CacheEntry concept

```
inline constexpr value_type const &get() const noexcept
```

Private Members

```
value_type value_
```

Friends

```
inline friend bool operator<(entry const &lhs, entry const &rhs)
    noexcept(noexcept(std::declval<value_type> const&>() <
                    std::declval<value_type> const&>()))
```

Forwarding operator< allowing to compare entries instead of the values.

hpx/cache/entries/fifo_entry.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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() = default

Any cache entry has to be default constructible.

inline explicit **fifo_entry**(*Value* const &val)
noexcept(*std*::is_nothrow_constructible_v<*base_type*, *Value* const&>)

Construct a new instance of a cache entry holding the given value.

inline explicit **fifo_entry**(*Value* &&val) noexcept

Construct a new instance of a cache entry holding the given value.

inline constexpr 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

Returns This function should return *true* if the entry should be added to the cache, otherwise it should return *false*.

inline constexpr *time_point* const &**get_creation_time()** const noexcept

Private Types

using **base_type** = *entry*<*Value*, *fifo_entry*<*Value*>>

using **time_point** = *std*::chrono::steady_clock::time_point

Private Members

time_point insertion_time_

Friends

inline friend bool **operator<**(*fifo_entry* const &lhs, *fifo_entry* const &rhs)
noexcept(noexcept(*std*::declval<*time_point* const&>() <
std::declval<*time_point* const&>()))

Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been created earlier (FIFO).

hpx/cache/entries/lfu_entry.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 `lru_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() = default

Any cache entry has to be default constructible.

```
inline explicit lfu_entry(Value const &val)  
    noexcept(std::is_nothrow_constructible_v<base_type, Value  
            const&>)
```

Construct a new instance of a cache entry holding the given value.

```
inline explicit lfu_entry(Value &&val) noexcept
```

Construct a new instance of a cache entry holding the given value.

inline bool **touch()** noexcept

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 will be used to compare the age of an entry during the invocation of the `operator<()`.

Returns This function should return true if the cache needs to update it's internal heap.

Usually this is needed if the entry has been changed by `touch()` in a way influencing the sort order as mandated by the cache's `UpdatePolicy`

```
inline constexpr unsigned long const &get_access_count() const noexcept
```

Private Types

```
using base_type = entry<Value, lfu_entry<Value>>
```

Private Members

```
unsigned long ref_count_ = 0
```

Friends

```
inline friend bool operator<(lfu_entry const &lhs, lfu_entry const &rhs) noexcept
    Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been accessed less frequently (LFU).
```

hpx/cache/entries/lru_entry.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

inline **lru_entry()**

Any cache entry has to be default constructible.

inline explicit **lru_entry**(*Value* const &val)
 noexcept(*std*::is_nothrow_constructible_v<*base_type*, *Value*
 const&>)

Construct a new instance of a cache entry holding the given value.

inline explicit **lru_entry**(*Value* &&val) noexcept

Construct a new instance of a cache entry holding the given value.

inline 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<()*.

Returns This function should return true if the cache needs to update it's internal heap.

Usually this is needed if the entry has been changed by *touch()* in a way influencing the sort order as mandated by the cache's UpdatePolicy

inline constexpr *time_point* const &**get_access_time()** const noexcept

Returns the last access time of the entry.

Private Types

using **base_type** = *entry*<*Value*, *lru_entry*<*Value*>>

using **time_point** = *std*::chrono::steady_clock::time_point

Private Members

time_point **access_time_**

Friends

inline friend bool **operator<(***lru_entry* const &lhs, *lru_entry* const &rhs)
 noexcept(noexcept(*std*::declval<*time_point* const&>() <
 std::declval<*time_point* const&>()))

Compare the 'age' of two entries. An entry is 'older' than another entry if it has been accessed less recently (LRU).

hpx/cache/entries/size_entry.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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() = default

Any cache entry has to be default constructible.

inline explicit size_entry(Value const &val, std::size_t size = 0)
noexcept(std::is_nothrow_constructible_v<base_type, Value const&>)

Construct a new instance of a cache entry holding the given value.

inline explicit size_entry(Value &&val, std::size_t size = 0) noexcept
 Construct a new instance of a cache entry holding the given value.

inline constexpr std::size_t get_size() const noexcept
 Return the ‘size’ of this entry.

Private Types

```
using derived_type = typename detail::size_derived<Value, Derived>::type  
  
using base_type = entry<Value, derived_type>
```

Private Members

```
std::size_t size_ = 0
```

Friends

```
inline friend constexpr friend bool operator< (size_entry const &lhs,  
size_entry const &rhs) noexcept  
Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has a bigger size.
```

hpx/cache/statistics/local_statistics.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **util**

namespace **cache**

namespace **statistics**

```
class local_statistics : public hpx::util::cache::statistics::no_statistics
```

Public Functions

```
local_statistics() = default  
  
inline std::size_t get_and_reset(std::size_t &value, bool reset) noexcept  
  
inline constexpr std::size_t hits() const noexcept  
  
inline constexpr std::size_t misses() const noexcept  
  
inline constexpr std::size_t insertions() const noexcept  
  
inline constexpr std::size_t evictions() const noexcept  
  
inline std::size_t hits(bool reset) noexcept
```

```

inline std::size_t misses(bool reset) noexcept
inline std::size_t insertions(bool reset) noexcept
inline std::size_t evictions(bool reset) noexcept
inline void got_hit() noexcept
    The function got_hit will be called by a cache instance whenever a entry got touched.
inline void got_miss() noexcept
    The function got_miss will be called by a cache instance whenever a requested entry has not
    been found in the cache.
inline void got_insertion() noexcept
    The function got_insertion will be called by a cache instance whenever a new entry has been
    inserted.
inline void got_eviction() noexcept
    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.
inline void clear() noexcept
    Reset all statistics.

```

Private Members

```

std::size_t hits_ = 0

std::size_t misses_ = 0

std::size_t insertions_ = 0

std::size_t evictions_ = 0

```

`hpx/cache/statistics/no_statistics.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

Defines

`HPX_CACHE_METHOD_UNSCOPED_ENUM_DEPRECATED_MSG`

namespace `hpx`

namespace `util`

namespace `cache`

namespace `statistics`

Enums

enum class **method**

Values:

enumerator **get_entry**

enumerator **insert_entry**

enumerator **update_entry**

enumerator **erase_entry**

Variables

constexpr *method* **method_get_entry** = *method*::*get_entry*

constexpr *method* **method_insert_entry** = *method*::*insert_entry*

constexpr *method* **method_update_entry** = *method*::*update_entry*

constexpr *method* **method_erase_entry** = *method*::*erase_entry*

class **no_statistics**

Subclassed by *hpx::util::cache::statistics::local_statistics*

Public Functions

inline constexpr void **got_hit()** const noexcept

The function *got_hit* will be called by a cache instance whenever a entry got touched.

inline constexpr void **got_miss()** const noexcept

The function *got_miss* will be called by a cache instance whenever a requested entry has not been found in the cache.

inline constexpr void **got_insertion()** const noexcept

The function *got_insertion* will be called by a cache instance whenever a new entry has been inserted.

inline constexpr void **got_eviction()** const noexcept

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.

inline constexpr void **clear()** const noexcept

Reset all statistics.

```
inline constexpr std::int64_t get_get_entry_count(bool) const noexcept
The function get_get_entry_count returns the number of invocations of the get_entry() API
function of the cache.

inline constexpr std::int64_t get_insert_entry_count(bool) const noexcept
The function get_insert_entry_count returns the number of invocations of the insert_entry()
API function of the cache.

inline constexpr std::int64_t get_update_entry_count(bool) const noexcept
The function get_update_entry_count returns the number of invocations of the update_entry()
API function of the cache.

inline constexpr std::int64_t get_erase_entry_count(bool) const noexcept
The function get_erase_entry_count returns the number of invocations of the erase() API
function of the cache.

inline constexpr std::int64_t get_get_entry_time(bool) const noexcept
The function get_get_entry_time returns the overall time spent executing of the get_entry()
API function of the cache.

inline constexpr std::int64_t get_insert_entry_time(bool) const noexcept
The function get_insert_entry_time returns the overall time spent executing of the in-
sert_entry() API function of the cache.

inline constexpr std::int64_t get_update_entry_time(bool) const noexcept
The function get_update_entry_time returns the overall time spent executing of the up-
date_entry() API function of the cache.

inline constexpr std::int64_t get_erase_entry_time(bool) const noexcept
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

```
inline constexpr update_on_exit(no_statistics const&, method) noexcept
```

compute_local

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/compute_local/vector.hpp

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

```
using value_type = T
Member types (FIXME: add reference to std.

using allocator_type = Allocator

using access_target = typename alloc_traits::access_target

using size_type = std::size_t

using difference_type = std::ptrdiff_t

using reference = typename alloc_traits::reference

using const_reference = typename alloc_traits::const_reference

using pointer = typename alloc_traits::pointer

using const_pointer = typename alloc_traits::const_pointer

using iterator = detail::iterator<T, Allocator>

using const_iterator = detail::iterator<T const, Allocator>

using reverse_iterator = detail::reverse_iterator<T, Allocator>

using const_reverse_iterator = detail::const_reverse_iterator<T, Allocator>
```

Public Functions

```

inline explicit vector(Allocator const &alloc = Allocator())

inline vector(size_type count, T const &value, Allocator const &alloc = Allocator())

inline explicit vector(size_type count, Allocator const &alloc = Allocator())

template<typename InIter, typename Enable = typename
         std::enable_if<hpx::traits::is_input_iterator<InIter>::value>::type>
inline vector(InIter first, InIter last, Allocator const &alloc)

inline vector(vector const &other)

inline vector(vector const &other, Allocator const &alloc)

inline vector(vector &&other)

inline vector(vector &&other, Allocator const &alloc)

inline vector(std::initializer_list<T> init, Allocator const &alloc)

inline ~vector()

inline vector &operator=(vector const &other)

inline vector &operator=(vector &&other)

inline allocator_type get_allocator() const noexcept
    Returns the allocator associated with the container.

inline reference operator[](size_type pos)

inline const_reference operator[](size_type pos) const

inline pointer data() noexcept
    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 deref-
    erenceable in that case).

inline const_pointer data() const noexcept
    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 deref-
    erenceable in that case).

inline T *device_data() const noexcept
    Returns a raw pointer corresponding to the address of the data allocated on the device.

inline std::size_t size() const noexcept

inline std::size_t capacity() const noexcept

inline bool empty() const noexcept
    Returns: size() == 0.

```

```
inline 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.

```
inline 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.

```
inline iterator begin() noexcept
```

```
inline iterator end() noexcept
```

```
inline const_iterator cbegin() const noexcept
```

```
inline const_iterator cend() const noexcept
```

```
inline const_iterator begin() const noexcept
```

```
inline const_iterator end() const noexcept
```

```
inline void swap(vector &other)
```

Effects: Exchanges the contents and capacity() of *this with that of x.

Complexity: Constant time.

```
inline void clear() noexcept
```

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_**

hpx/compute_local/host/block_executor.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
template<typename Executor>
struct
  hpx::parallel::execution::executor_execution_category<compute::host::block_executor<Executor>>
```

Public Types

```
typedef hpx::execution::parallel_execution_tag type

template<typename Executor>
struct is_one_way_executor<compute::host::block_executor<Executor>> : public true_type

template<typename Executor>
struct is_two_way_executor<compute::host::block_executor<Executor>> : public true_type

template<typename Executor>
struct is_bulk_one_way_executor<compute::host::block_executor<Executor>> : public true_type

template<typename Executor>
struct is_bulk_two_way_executor<compute::host::block_executor<Executor>> : public true_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

```
using executor_parameters_type = hpx::execution::static_chunk_size
```

Public Functions

```
inline explicit 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 = {})

inline explicit block_executor(std::vector<host::target> &&targets)

inline block_executor(block_executor const &other)

inline block_executor(block_executor &&other) noexcept

inline block_executor &operator=(block_executor const &other)

inline block_executor &operator=(block_executor &&other) noexcept

inline std::vector<host::target> const &targets() const
```

Private Functions

```
inline auto get_next_executor() const

template<typename F, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::post_t, block_executor const
                                       &exec, F &&f, Ts&&... ts)

template<typename F, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::async_execute_t,
                                       block_executor const &exec, F &&f, Ts&&... ts)

template<typename F, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::sync_execute_t,
                                       block_executor const &exec, F &&f, Ts&&... ts)

template<typename F, typename Shape, typename ...Ts>
inline decltype(auto) bulk_async_execute_impl(F &&f, Shape const &shape, Ts&&... ts)
                                              const

template<typename F, typename Shape, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::bulk_async_execute_t,
                                       block_executor const &exec, F &&f, Shape const
                                       &shape, Ts&&... ts)

template<typename F, typename Shape, typename ...Ts>
inline decltype(auto) bulk_sync_execute_impl(F &&f, Shape const &shape, Ts&&... ts)
                                              const

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

```
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::bulk_sync_execute_t,
                                         block_executor const &exec, F &&f, Shape const
                                         &shape, Ts&&... ts)
```

```
inline void init_executors()
```

Private Members

```
std::vector<host::target> targets_
```

```
mutable 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> block_executor< Executor > >
```

Public Types

```
typedef hpx::execution::parallel_execution_tag type
```

```
template<typename Executor> block_executor< Executor > > : public true_type
```

```
template<typename Executor> block_executor< Executor > > : public true_type
```

```
template<typename Executor> block_executor< Executor > > : public true_type
```

```
template<typename Executor> block_executor< Executor > > : public true_type
```

hpx/compute_local/host/block_fork_join_executor.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **execution**

namespace **experimental**

class **block_fork_join_executor**

#include <block_fork_join_executor.hpp> An executor with fork-join (blocking) semantics.

The `block_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`.

This behaviour is similar to the plain `fork_join_executor` except that the `block_fork_join_executor` creates a hierarchy of `fork_join_executors`, one for each target used to initialize it.

Public Functions

```
inline explicit block_fork_join_executor(threads::thread_priority priority =  
    threads::thread_priority::bound,  
    threads::thread_stacksize stacksize =  
    threads::thread_stacksize::small_,  
    fork_join_executor::loop_schedule schedule =  
    fork_join_executor::loop_schedule::static_,  
    std::chrono::nanoseconds yield_delay =  
    std::chrono::milliseconds(1))
```

Construct a *block_fork_join_executor*.

Note: This constructor will create one fork_join_executor for each numa domain

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.

```
inline explicit block_fork_join_executor(std::vector<compute::host::target> const
&targets, threads::thread_priority priority =
threads::thread_priority::bound,
threads::thread_stacksize stacksize =
threads::thread_stacksize::small_,
fork_join_executor::loop_schedule schedule =
fork_join_executor::loop_schedule::static_,
std::chrono::nanoseconds yield_delay =
std::chrono::milliseconds(1))
```

Construct a *block_fork_join_executor*.

Note: This constructor will create one fork_join_executor for each given target

Parameters

- **targets** – The list of targets to use for thread placement
- **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.

```
template<typename F, typename S, typename ...Ts>
inline void bulk_sync_execute_helper(F &&f, S const &shape, Ts&&... ts)
```

```
template<typename F, typename S, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::bulk_async_execute_t,
block_fork_join_executor &exec, F &&f, S const
&shape, Ts&&... ts)
```

```
template<typename Tag>
inline decltype(auto) friend tag_invoke(Tag tag, block_fork_join_executor const &exec)
noexcept
```

Private Members

fork_join_executor **exec_**

std::vector<fork_join_executor> **block_execs_**

Private Static Functions

```
static inline hpx::threads::mask_type cores_for_targets(std::vector<compute::host::target>
const &targets)
```

Friends

```
template<typename F, typename S, typename ...Ts>
inline friend void tag_invoke(hpx::parallel::execution::bulk_sync_execute_t,
                            block_fork_join_executor &exec, F &&f, S const &shape,
                            Ts&&... ts)

template<typename Tag, typename Property>
inline friend block_fork_join_executor tag_invoke(Tag tag, block_fork_join_executor const
                                                &exec, Property &&prop) noexcept
```

namespace **parallel**

namespace **execution**

config

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/config/endian.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

coroutines

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/coroutines/thread_enums.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

namespace **threads**

Enums

enum class **thread_schedule_state** : *std::int8_t*

The *thread_schedule_state* enumerator encodes the current state of a *thread* instance

Values:

enumerator **unknown**

enumerator **active**

thread is currently active (running, has resources)

enumerator pending

thread is pending (ready to run, but no hardware resource available)

enumerator suspended

thread has been suspended (waiting for synchronization event, but still known and under control of the thread-manager)

enumerator depleted

thread has been depleted (deeply suspended, it is not known to the thread-manager)

enumerator terminated

thread has been stopped an may be garbage collected

enumerator staged

this is not a real thread state, but allows to reference staged task descriptions, which eventually will be converted into thread objects

enumerator pending_do_not_schedule**enumerator pending_boost****enum class `thread_priority` : `std::int8_t`**

This enumeration lists all possible thread-priorities for HPX threads.

Values:

enumerator unknown**enumerator default_**

Will assign the priority of the task to the default (normal) priority.

enumerator low

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.

enumerator normal

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.

enumerator high_recursive

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.

enumerator boost

Same as `thread_priority_high` except that the thread will fall back to `thread_priority_normal` if resumed after being suspended.

enumerator `high`

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).

enumerator `bound`

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 class `thread_restart_state` : `std::int8_t`

The `thread_restart_state` enumerator encodes the reason why a thread is being restarted

Values:

enumerator `unknown`**enumerator `signaled`**

The thread has been signaled.

enumerator `timeout`

The thread has been reactivated after a timeout

enumerator `terminate`

The thread needs to be terminated.

enumerator `abort`

The thread needs to be aborted.

enum class `thread_stacksize` : `std::int8_t`

A `thread_stacksize` references any of the possible stack-sizes for HPX threads.

Values:

enumerator `unknown`**enumerator `small_`**

use small stack size (the underscore is to work around small being defined to char on Windows)

enumerator `medium`

use medium sized stack size

enumerator `large`

use large stack size

enumerator `huge`

use very large stack size

enumerator `nostack`

this thread does not suspend (does not need a stack)

enumerator `current`

use size of current thread's stack

enumerator `default_`

use default stack size

enumerator `minimal`

use minimally stack size

enumerator `maximal`

use maximally stack size

enum class `thread_schedule_hint_mode` : `std::int8_t`

The type of hint given when creating new tasks.

*Values:***enumerator `none`**

A hint that leaves the choice of scheduling entirely up to the scheduler.

enumerator `thread`

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.

enumerator `numa`

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

```
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

```
inline constexpr thread_schedule_hint() noexcept
```

Construct a default hint with mode `thread_schedule_hint_mode::none`.

```
inline explicit constexpr thread_schedule_hint(std::int16_t thread_hint) noexcept
```

Construct a hint with mode `thread_schedule_hint_mode::thread` and the given hint as the local thread number.

```
inline constexpr thread_schedule_hint(thread_schedule_hint_mode mode, std::int16_t hint)  
noexcept
```

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.

hpx/coroutines/thread_id_type.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

template<>

```
struct std::hash<::hpx::threads::thread\_id>
```

Public Functions

```
inline std::size\_t operator\(\) \(::hpx::threads::thread\_id const &v\) const noexcept
```

template<>

```
struct std::hash<::hpx::threads::thread\_id\_ref>
```

Public Functions

```
inline std::size\_t operator\(\) \(::hpx::threads::thread\_id\_ref const &v\) const noexcept
```

namespace **hpx**

 namespace **threads**

Enums

```
enum class thread_id_addref
```

Values:

enumerator **yes**

enumerator **no**

Variables

```
constexpr const thread\_id invalid\_thread\_id
```

```
struct thread_id
```

Public Functions

```
thread_id() noexcept = default  
thread_id(thread_id const&) = default  
thread_id &operator=(thread_id const&) = default  
inline constexpr thread_id(thread_id &&rhs) noexcept  
inline constexpr thread_id &operator=(thread_id &&rhs) noexcept  
inline explicit constexpr thread_id(thread_id_repr const &thrd) noexcept  
inline constexpr thread_id &operator=(thread_id_repr const &rhs) noexcept  
inline explicit constexpr operator bool() const noexcept  
inline constexpr thread_id_repr get() const noexcept  
inline constexpr void reset() noexcept
```

Private Types

```
using thread_id_repr = void*
```

Private Members

```
thread_id_repr thrd_ = nullptr
```

Friends

```
inline friend constexpr friend bool operator==(std::nullptr_t,  
thread_id const &rhs) noexcept  
inline friend constexpr friend bool operator!=(std::nullptr_t,  
thread_id const &rhs) noexcept  
inline friend constexpr friend bool operator==(thread_id const &lhs,  
std::nullptr_t) noexcept  
inline friend constexpr friend bool operator!=(thread_id const &lhs,  
std::nullptr_t) noexcept  
inline friend constexpr friend bool operator==(thread_id const &lhs,  
thread_id const &rhs) noexcept  
inline friend constexpr friend bool operator!=(thread_id const &lhs,  
thread_id const &rhs) noexcept
```

```

inline friend constexpr friend bool operator< (thread_id const &lhs,
thread_id const &rhs) noexcept

inline friend constexpr friend bool operator> (thread_id const &lhs,
thread_id const &rhs) noexcept

inline friend constexpr friend bool operator<= (thread_id const &lhs,
thread_id const &rhs) noexcept

inline friend constexpr friend bool operator>= (thread_id const &lhs,
thread_id const &rhs) noexcept

template<typename Char, typename Traits>
inline friend std::basic_ostream<Char, Traits> &operator<<(std::basic_ostream<Char, Traits>
&os, thread_id const &id)

inline friend 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() noexcept = default

thread_id_ref(thread_id_ref const&) = default

thread_id_ref &operator=(thread_id_ref const&) = default

thread_id_ref(thread_id_ref &&rhs) noexcept = default

thread_id_ref &operator=(thread_id_ref &&rhs) noexcept = default

inline explicit thread_id_ref(thread_id_repr const &thrd) noexcept

inline explicit thread_id_ref(thread_id_repr &&thrd) noexcept

inline thread_id_ref &operator=(thread_id_repr const &rhs) noexcept

inline thread_id_ref &operator=(thread_id_repr &&rhs) noexcept

inline explicit thread_id_ref(thread_repr *thrd, thread_id_addrref addref =
thread_id_addrref::yes) noexcept

inline thread_id_ref &operator=(thread_repr *rhs) noexcept

inline thread_id_ref(thread_id const &noref)

```

```
inline thread_id_ref(thread_id &&noref) noexcept
inline thread_id_ref &operator=(thread_id const &noref)
inline thread_id_ref &operator=(thread_id &&noref) noexcept
inline explicit operator bool() const noexcept
inline thread_id noref() const noexcept
inline thread_id_repr &get() & noexcept
inline thread_id_repr &&get() && noexcept
inline thread_id_repr const &get() const & noexcept
inline void reset() noexcept
inline void reset(thread_repr *thrd, bool add_ref = true) noexcept
inline constexpr thread_repr *detach() noexcept
```

Private Types

```
using thread_id_repr = hpx::intrusive_ptr<detail::thread_data_reference_counting>
```

Private Members

```
thread_id_repr thrd_
```

Friends

```
inline friend bool operator==(std::nullptr_t, thread_id_ref const &rhs) noexcept
inline friend bool operator!=(std::nullptr_t, thread_id_ref const &rhs) noexcept
inline friend bool operator==(thread_id_ref const &lhs, std::nullptr_t) noexcept
inline friend bool operator!=(thread_id_ref const &lhs, std::nullptr_t) noexcept
inline friend bool operator==(thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept
inline friend bool operator!=(thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept
inline friend bool operator<(thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept
inline friend bool operator>(thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept
inline friend bool operator<=(thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept
inline friend bool operator>=(thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept
template<typename Char, typename Traits>
```

```
inline friend std::basic_ostream<Char, Traits> &operator<<(std::basic_ostream<Char, Traits>
    &os, thread_id_ref const &id)

inline friend void format_value(std::ostream &os, boost::string_ref spec, thread_id_ref const
    &id)

namespace std

template<> thread_id >
```

Public Functions

```
inline std::size_t operator()(::hpx::threads::thread_id const &v) const noexcept
```

```
template<> thread_id_ref >
```

Public Functions

```
inline std::size_t operator()(::hpx::threads::thread_id_ref const &v) const noexcept
```

datastructures

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/datastructures/any.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
template<>
```

```
class hpx::util::basic_any<void, void, void, std::true_type>
```

Public Functions

```
inline constexpr basic_any() noexcept
```

```
inline basic_any(basic_any const &x)
```

```
inline basic_any(basic_any &&x) noexcept
```

```
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename
    std::decay<T>::type>::value>::type>
```

```
inline basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename
    std::decay<T>::type>::value>::type* = nullptr)
```

```
template<typename T, typename ...Ts, typename Enable = typename
    std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value &&
    std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
```

```
inline explicit 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, Ts...>::value &&  
std::is_copy_constructible<typename std::decay<T>::type>::value>::type>  
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)  
  
inline ~basic_any()  
  
inline basic_any &operator=(basic_any const &x)  
  
inline basic_any &operator=(basic_any &&rhs) noexcept  
  
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename  
std::decay<T>::type>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>  
inline basic_any &operator=(T &&rhs)  
  
inline basic_any &swap(basic_any &x) noexcept  
  
inline std::type_info const &type() const  
  
template<typename T>  
inline T const &cast() const  
  
inline bool has_value() const noexcept  
  
inline void reset()  
  
inline bool equal_to(basic_any const &rhs) const noexcept
```

Private Functions

```
inline 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 inline void new_object(void *&object, std::true_type, Ts&&... ts)
```

```
template<typename T, typename ...Ts>  
static inline void new_object(void *&object, std::false_type, Ts&&... ts)
```

```
template<typename Char>
```

```
class hpx::util::basic_any<void, void, Char, std::true_type>
```

Public Functions

```

inline constexpr basic_any() noexcept

inline basic_any(basic_any const &x)

inline basic_any(basic_any &&x) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
inline basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::type>::value* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit 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, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

inline ~basic_any()

inline basic_any &operator=(basic_any const &x)

inline basic_any &operator=(basic_any &&rhs) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline basic_any &operator=(T &&rhs) noexcept

inline basic_any &swap(basic_any &x) noexcept

inline std::type_info const &type() const

template<typename T>
inline T const &cast() const

inline bool has_value() const noexcept

inline void reset()

inline bool equal_to(basic_any const &rhs) const noexcept

```

Private Functions

```
inline 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 inline void new_object(void *&object, std::true_type, Ts&&... ts)  
  
template<typename T, typename ...Ts>  
static inline void new_object(void *&object, std::false_type, Ts&&... ts)  
  
template<>  
class hpx::util::basic_any<void, void, void, std::false_type>
```

Public Functions

```
inline constexpr basic_any() noexcept  
inline basic_any(basic_any &&x) noexcept  
  
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>  
inline basic_any(T &&x, typename std::enable_if<std::is_move_constructible<typename std::decay<T>::value>::type* = nullptr)<br/>  
  
template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::value>::type>  
inline explicit basic_any(std::in_place_type_t<T>, Ts&&... ts)<br/>  
  
template<typename T, typename U, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::value>::type>  
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)<br/>  
  
basic_any(basic_any const &x) = delete  
basic_any &operator=(basic_any const &x) = delete  
  
inline ~basic_any()  
  
inline basic_any &operator=(basic_any &&rhs) noexcept  
  
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value && std::is_move_constructible<typename std::decay<T>::value>::type>  
inline basic_any &operator=(T &&rhs)<br/>  
  
inline basic_any &swap(basic_any &x) noexcept
```

```
inline std::type_info const &type() const
template<typename T>
inline T const &cast() const
inline bool has_value() const noexcept
inline void reset()
inline bool equal_to(basic_any const &rhs) const noexcept
```

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 inline void new_object(void *&object, std::true_type, Ts&&... ts)
template<typename T, typename ...Ts>
static inline void new_object(void *&object, std::false_type, Ts&&... ts)
template<typename Char>
class hpx::util::basic_any<void, void, Char, std::false_type>
```

Public Functions

```
inline constexpr basic_any() noexcept
inline basic_any(basic_any &&x) noexcept
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
inline basic_any(T &&x, typename std::enable_if<std::is_move_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>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::value>::type>
inline explicit 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, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::value>::type>
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)
basic_any(basic_any const &x) = delete
```

```
basic_any &operator=(basic_any const &x) = delete
inline ~basic_any()
inline basic_any &operator=(basic_any &&rhs) noexcept
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value && std::is_move_constructible<typename std::decay<T>::type>::value>::type>
inline basic_any &operator=(T &&rhs) noexcept
inline basic_any &swap(basic_any &x) noexcept
inline std::type_info const &type() const
template<typename T>
inline T const &cast() const
inline bool has_value() const noexcept
inline void reset()
inline bool equal_to(basic_any const &rhs) const noexcept
```

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 inline void new_object(void *&object, std::true_type, Ts&&... ts)
template<typename T, typename ...Ts>
static inline void new_object(void *&object, std::false_type, Ts&&... ts)
```

namespace **hpx**

Top level HPX namespace.

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,
T 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>
inline T *any_cast(util::basic_any<IArch, OArch, Char, Copyable> *operand) noexcept

Performs type-safe access to the contained object.

Parameters **operand** – target any object

Returns If *operand* is not a null pointer, and the *typeid* of the requested *T* matches that of the contents of *operand*, a pointer to the value contained by *operand*, otherwise a null pointer.

template<typename T, typename IArch, typename OArch, typename Char, typename Copyable>
inline T const *any_cast(util::basic_any<IArch, OArch, Char, Copyable> const *operand) noexcept

Performs type-safe access to the contained object.

Parameters **operand** – target any object

Returns If *operand* is not a null pointer, and the *typeid* of the requested *T* matches that of the contents of *operand*, a pointer to the value contained by *operand*, otherwise a null pointer.

template<typename T, typename IArch, typename OArch, typename Char, typename Copyable>
T any_cast(util::basic_any<IArch, OArch, Char, Copyable> &operand)

Performs type-safe access to the contained object. Let *U* be *std::remove_cv_t<std::remove_reference_t<T>>* The program is ill-formed if *std::is_constructible_v<T, U&>* is false.

Parameters **operand** – target any object

Returns static_cast<T>(*std::any_cast<U>(&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)

Performs type-safe access to the contained object. Let *U* be *std::remove_cv_t<std::remove_reference_t<T>>* The program is ill-formed if *std::is_constructible_v<T, const U&>* is false.

Parameters **operand** – target any object

Returns static_cast<T>(*std::any_cast<U>(&operand))

```
struct bad_any_cast : public bad_cast
```

#include <any.hpp> Defines a type of object to be thrown by the value-returning forms of `hpx::any_cast` on failure.

Public Functions

```
inline bad_any_cast(std::type_info const &src, std::type_info const &dest)
```

Constructs a new `bad_any_cast` object with an implementation-defined null-terminated byte string which is accessible through `what()`.

```
inline const char *what() const noexcept override
```

Returns the explanatory string.

Note: Implementations are allowed but not required to override `what()`.

Returns Pointer to a null-terminated string with explanatory information. The string is suitable for conversion and display as a `std::wstring`. The pointer is guaranteed to be valid at least until the exception object from which it is obtained is destroyed, or until a non-const member function (e.g. copy assignment operator) on the exception object is called.

Public Members

```
const char *from
```

```
const char *to
```

```
namespace util
```

Typedefs

```
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::enable_if<!std::is_void<Char>::value>::type>
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::enable_if<!std::is_void<Char>::value>::type>
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 Copyableswap(basic_any<IArch, OArch, Char, Copyable> &lhs, basic_any<IArch, OArch, Char, Copyable>
&rhs) noexcept

template<typename T, typename Char, typename ...Ts>
basic_any<void, void, Char, std::true_type> make_streamable_any_nonser(T&&... 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 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, typename Char>
basic_any<void, void, Char, std::true_type> make_streamable_any_nonser(T &&t)

template<typename T, typename Char>
basic_any<void, void, Char, std::false_type> make_streamable_unique_any_nonser(T &&t)

template<typename IArch, typename OArch, typename Char = char, typename Copyable =
std::true_type>
class basic_any

template<typename Char> false_type >

```

Public Functions

```

inline constexpr basic_any() noexcept

inline basic_any(basic_any &&x) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,
std::decay<T>::type>::value>::type>
inline basic_any(T &&x, typename std::enable_if<std::is_move_constructible<typename
std::decay<T>::type>::value>::type* = nullptr)

```

```
template<typename T, typename ...Ts, typename Enable = typename
        std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value &&
        std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit 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, Ts...>::value &&
        std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

basic_any(basic_any const &x) = delete
basic_any &operator=(basic_any const &x) = delete
inline ~basic_any()

inline basic_any &operator=(basic_any &&rhs) noexcept
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,
        typename std::decay<T>::type>::value && std::is_move_constructible<typename
        std::decay<T>::type>::value>::type>
inline basic_any &operator=(T &&rhs) noexcept
inline basic_any &swap(basic_any &x) noexcept
inline std::type_info const &type() const
template<typename T>
inline T const &cast() const
inline bool has_value() const noexcept
inline void reset()
inline bool equal_to(basic_any const &rhs) const noexcept
```

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 inline void new_object(void *&object, std::true_type, Ts&&... ts)
template<typename T, typename ...Ts>
static inline void new_object(void *&object, std::false_type, Ts&&... ts)
```

```
template<typename Char> true_type >
```

Public Functions

```

inline constexpr basic_any() noexcept

inline basic_any(basic_any const &x)

inline basic_any(basic_any &&x) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,
typename std::decay<T>::type>::value>::type>
inline basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename
std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename
std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value &&
std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit 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, Ts...>::value &&
std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

inline ~basic_any()

inline basic_any &operator=(basic_any const &x)

inline basic_any &operator=(basic_any &&rhs) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,
typename std::decay<T>::type>::value && std::is_copy_constructible<typename
std::decay<T>::type>::value>::type>
inline basic_any &operator=(T &&rhs) noexcept

inline basic_any &swap(basic_any &x) noexcept

inline std::type_info const &type() const

template<typename T>
inline T const &cast() const

inline bool has_value() const noexcept

inline void reset()

inline bool equal_to(basic_any const &rhs) const noexcept

```

Private Functions

```
inline 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 inline void new_object(void *&object, std::true_type, Ts&&... ts)
```

```
template<typename T, typename ...Ts>  
static inline void new_object(void *&object, std::false_type, Ts&&... ts)
```

```
template<> false_type >
```

Public Functions

```
inline constexpr basic_any() noexcept
```

```
inline basic_any(basic_any &&x) noexcept
```

```
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,  
typename std::decay<T>::type>::value>::type>  
inline basic_any(T &&x, typename std::enable_if<std::is_move_constructible<typename  
std::decay<T>::type>::value>::type* = nullptr)
```

```
template<typename T, typename ...Ts, typename Enable = typename  
std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value &&  
std::is_copy_constructible<typename std::decay<T>::type>::value>::type>  
inline explicit 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, Ts...>::value &&  
std::is_copy_constructible<typename std::decay<T>::type>::value>::type>  
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)
```

```
basic_any(basic_any const &x) = delete
```

```
basic_any &operator=(basic_any const &x) = delete
```

```
inline ~basic_any()
```

```
inline basic_any &operator=(basic_any &&rhs) noexcept
```

```
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,  
typename std::decay<T>::type>::value && std::is_move_constructible<typename  
std::decay<T>::type>::value>::type>  
inline basic_any &operator=(T &&rhs)
```

```
inline basic_any &swap(basic_any &x) noexcept
```

```
inline std::type_info const &type() const
template<typename T>
inline T const &cast() const
inline bool has_value() const noexcept
inline void reset()
inline bool equal_to(basic_any const &rhs) const noexcept
```

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 inline void new_object(void *&object, std::true_type, Ts&&... ts)
template<typename T, typename ...Ts>
static inline void new_object(void *&object, std::false_type, Ts&&... ts)

template<> true_type >
```

Public Functions

```
inline constexpr basic_any() noexcept
inline basic_any(basic_any const &x)
inline basic_any(basic_any &&x) noexcept
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
inline basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::type>::value>::type* = nullptr)
template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit 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, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)
inline ~basic_any()
```

```
inline basic_any &operator=(basic_any const &x)  
inline basic_any &operator=(basic_any &&rhs) noexcept  
  
template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,  
typename std::decay<T>::type>::value && std::is_copy_constructible<typename  
std::decay<T>::type>::value>::type>  
inline basic_any &operator=(T &&rhs)  
  
inline basic_any &swap(basic_any &x) noexcept  
inline std::type_info const &type() const  
  
template<typename T>  
inline T const &cast() const  
  
inline bool has_value() const noexcept  
inline void reset()  
inline bool equal_to(basic_any const &rhs) const noexcept
```

Private Functions

```
inline 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 inline void new_object(void *&object, std::true_type, Ts&&... ts)  
  
template<typename T, typename ...Ts>  
static inline void new_object(void *&object, std::false_type, Ts&&... ts)
```

hpx/datastructures/tuple.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace hpx

Top level HPX namespace.

Functions

```
template<typename ...Ts>
constexpr tuple<util::decay_unwrap_t<Ts>...> make_tuple(Ts&&... ts)
```

Provides compile-time indexed access to the types of the elements of the tuple.

```
template<typename ...Ts>
constexpr tuple<Ts&&...> forward_as_tuple(Ts&&... ts)
```

Constructs a tuple of references to the arguments in args suitable for forwarding as an argument to a function. The tuple has rvalue reference data members when rvalues are used as arguments, and otherwise has lvalue reference data members.

Parameters **ts** – zero or more arguments to construct the tuple from

Returns *hpx::tuple* object created as if by

```
hpx::tuple<Ts&&...>(HPX_FORWARD(Ts, ts)...)
```

```
template<typename ...Ts>
constexpr tuple<Ts&&...> tie(Ts&... ts)
```

Creates a tuple of lvalue references to its arguments or instances of *hpx::ignore*.

Parameters **ts** – zero or more lvalue arguments to construct the tuple from.

Returns *hpx::tuple* object containing lvalue references.

```
template<typename ...Tuples>
constexpr auto tuple_cat(Tuples&&... tuples)
```

Constructs a tuple that is a concatenation of all tuples in *tuples*. The behavior is undefined if any type in `std::decay_t<Tuples>...` is not a specialization of *hpx::tuple*. However, an implementation may choose to support types (such as `std::array` and `std::pair`) that follow the tuple-like protocol.

Parameters **tuples** -- zero or more tuples to concatenate

Returns *hpx::tuple* object composed of all elements of all argument tuples constructed from `hpx::get<Is>(HPX_FORWARD(UTuple,t))` for each individual element.

```
template<std::size_t I>
util::at_index<I, Ts...>::type &get() noexcept
```

Extracts the Ith element from the tuple. I must be an integer value in [0, `sizeof...(Ts)`).

```
template<std::size_t I>
util::at_index<I, Ts...>::type const &get() const noexcept
```

Extracts the Ith element from the tuple. I must be an integer value in [0, `sizeof...(Ts)`).

Variables

```
constexpr hpx::detail::ignore_type ignore = {}
```

An object of unspecified type such that any value can be assigned to it with no effect. Intended for use with *hpx::tie* when unpacking a *hpx::tuple*, as a placeholder for the arguments that are not used. While the behavior of *hpx::ignore* outside of *hpx::tie* is not formally specified, some code guides recommend using *hpx::ignore* to avoid warnings from unused return values of `[[nodiscard]]` functions.

```
template<typename ...Ts>
```

class tuple

#include <tuple.hpp> Class template `hpx::tuple` is a fixed-size collection of heterogeneous values. It is a generalization of `hpx::pair`. If `std::is_trivially_destructible<Ti>::value` is true for every `Ti` in `Ts`, the destructor of tuple is trivial.

Param `Ts...` the types of the elements that the tuple stores.

template<`std::size_t` `I`, typename `T`, typename `Enable` = void>

struct tuple_element

#include <tuple.hpp> Provides compile-time indexed access to the types of the elements of a tuple-like type.

The primary template is not defined. An explicit (full) or partial specialization is required to make a type tuple-like.

template<typename `T`>

struct tuple_size

#include <tuple.hpp> Provides access to the number of elements in a tuple-like type as a compile-time constant expression.

The primary template is not defined. An explicit (full) or partial specialization is required to make a type tuple-like.

hpx/datastructures/serialization/serializable_any.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

template<typename `IArch`, typename `OArch`, typename `Char`>

class `hpx::util::basic_any<IArch, OArch, Char, std::true_type>`

Public Functions

inline constexpr `basic_any()` noexcept

inline `basic_any(basic_any const &x)`

inline `basic_any(basic_any &&x)` noexcept

template<typename `T`, typename `Enable` = typename `std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>`

inline `basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::type>::value*>::type* = nullptr)`

template<typename `T`, typename ...`Ts`, typename `Enable` = typename `std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>`

inline explicit `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, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>`

```

inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

inline ~basic_any()

inline basic_any &operator=(basic_any const &x)

inline basic_any &operator=(basic_any &&rhs) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline basic_any &operator=(T &&rhs)

inline basic_any &swap(basic_any &x) noexcept

inline std::type_info const &type() const

template<typename T>
inline T const &cast() const

inline bool has_value() const noexcept

inline void reset()

inline bool equal_to(basic_any const &rhs) const noexcept

```

Private Functions

```

inline basic_any &assign(basic_any const &x)

inline void load(IArch &ar, const unsigned version)

inline 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 inline void new_object(void *&object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>
static inline void new_object(void *&object, std::false_type, Ts&&... ts)

```

Friends

```
friend class hpx::serialization::access
```

namespace **hpx**

Top level HPX namespace.

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)
```

Constructs an any object containing an object of type *T*, passing the provided arguments to *T*'s constructor.
Equivalent to:

```
return std::any(std::in_place_type<T>, std::forward<Args>(args)...);
```

namespace **util**

TypeDefs

```
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 IArch, typename OArch, typename Char> true_type >
```

Public Functions

```

inline constexpr basic_any() noexcept

inline basic_any(basic_any const &x)

inline basic_any(basic_any &&x) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,
typename std::decay<T>::type>::value>::type>
inline basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename
std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename
std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value &&
std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit 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, Ts...>::value &&
std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
inline explicit basic_any(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

inline ~basic_any()

inline basic_any &operator=(basic_any const &x)

inline basic_any &operator=(basic_any &&rhs) noexcept

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any,
typename std::decay<T>::type>::value && std::is_copy_constructible<typename
std::decay<T>::type>::value>::type>
inline basic_any &operator=(T &&rhs)

inline basic_any &swap(basic_any &x) noexcept

inline std::type_info const &type() const

template<typename T>
inline T const &cast() const

inline bool has_value() const noexcept

inline void reset()

inline bool equal_to(basic_any const &rhs) const noexcept

```

Private Functions

```

inline basic_any &assign(basic_any const &x)

inline void load(IArch &ar, const unsigned version)

inline 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 inline void new_object(void *&object, std::true_type, Ts&&... ts)  
  
template<typename T, typename ...Ts>  
static inline void new_object(void *&object, std::false_type, Ts&&... ts)
```

Friends

```
friend class hpx::serialization::access  
  
struct hash_any
```

Public Functions

```
template<typename Char>  
std::size_t operator()(const basic_any<serialization::input_archive, serialization::output_archive,> &elem) const
```

debugging

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

hpx/debugging/print.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_DP_LAZY(Expr, printer)

errors

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/errors/error.hpp

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:

enumerator **success**

The operation was successful.

enumerator **no_success**

The operation did failed, but not in an unexpected manner.

enumerator **not_implemented**

The operation is not implemented.

enumerator **out_of_memory**

The operation caused an out of memory condition.

enumerator **bad_action_code**

enumerator **bad_component_type**

The specified component type is not known or otherwise invalid.

enumerator **network_error**

A generic network error occurred.

enumerator **version_too_new**

The version of the network representation for this object is too new.

enumerator **version_too_old**

The version of the network representation for this object is too old.

enumerator `version_unknown`

The version of the network representation for this object is unknown.

enumerator `unknown_component_address`**enumerator `duplicate_component_address`**

The given global id has already been registered.

enumerator `invalid_status`

The operation was executed in an invalid status.

enumerator `bad_parameter`

One of the supplied parameters is invalid.

enumerator `internal_server_error`**enumerator `service_unavailable`****enumerator `bad_request`****enumerator `repeated_request`****enumerator `lock_error`****enumerator `duplicate_console`**

There is more than one console locality.

enumerator `no_registered_console`

There is no registered console locality available.

enumerator `startup_timed_out`**enumerator `uninitialized_value`****enumerator `bad_response_type`****enumerator `deadlock`****enumerator `assertion_failure`****enumerator `null_thread_id`**

Attempt to invoke a API function from a non-HPX thread.

enumerator **invalid_data**

enumerator **yield_aborted**

The yield operation was aborted.

enumerator **dynamic_link_failure**

enumerator **commandline_option_error**

One of the options given on the command line is erroneous.

enumerator **serialization_error**

There was an error during serialization of this object.

enumerator **unhandled_exception**

An unhandled exception has been caught.

enumerator **kernel_error**

The OS kernel reported an error.

enumerator **broken_task**

The task associated with this future object is not available anymore.

enumerator **task_moved**

The task associated with this future object has been moved.

enumerator **task_already_started**

The task associated with this future object has already been started.

enumerator **future_already_retrieved**

The future object has already been retrieved.

enumerator **promise_already_satisfied**

The value for this future object has already been set.

enumerator **future_does_not_support_cancellation**

The future object does not support cancellation.

enumerator **future_can_not_be_cancelled**

The future can't be canceled at this time.

enumerator **no_state**

The future object has no valid shared state.

enumerator **broken.promise**

The promise has been deleted.

enumerator **thread_resource_error**

enumerator **future_cancelled**

enumerator **thread_cancelled**

enumerator **thread_not_interruptable**

enumerator **duplicate_component_id**

The component type has already been registered.

enumerator **unknown_error**

An unknown error occurred.

enumerator **bad_plugin_type**

The specified plugin type is not known or otherwise invalid.

enumerator **filesystem_error**

The specified file does not exist or other filesystem related error.

enumerator **bad_function_call**

equivalent of std::bad_function_call

enumerator **task_canceled_exception**

parallel::v2::task_canceled_exception

enumerator **task_block_not_active**

task_region is not active

enumerator **out_of_range**

Equivalent to std::out_of_range.

enumerator **length_error**

Equivalent to std::length_error.

enumerator **migration_needs_retry**

migration failed because of global race, retry

hpx/errors/error_code.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Unnamed Group

inline `error_code make_error_code(error e, throwmode mode = throwmode::plain)`

Returns a new `error_code` constructed from the given parameters.

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

inline `error_code make_error_code(error e, char const *msg, throwmode mode = throwmode::plain)`

Returns `error_code(e, msg, mode)`.

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

inline `error_code make_error_code(error e, std::string const &msg, throwmode mode = throwmode::plain)`
Returns `error_code(e, msg, mode)`.

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

inline `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.

inline `error_code make_success_code(throwmode mode = throwmode::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

inline explicit **error_code**(*throwmode* mode = *throwmode::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*).

Throws nothing –

explicit **error_code**(*error e, throwmode* mode = *throwmode::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*).

Throws nothing –

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

Throws nothing –

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

Throws *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 = *throwmode::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*).

Throws *std::bad_alloc* – (if allocation of a copy of the passed string fails).

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

Throws *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* = *throwmode::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*).

Throws *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*.

Throws nothing –

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

Private Members

```
std::exception_ptr exception_
```

Friends

```
friend class exception  
friend error_code make_error_code(std::exception_ptr const&)
```

hpx/errors/exception.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

See also:

```
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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns The error message stored in the exception If the exception instance does not hold this information, the function will return an empty string.

`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.

See also:

`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`.

Throws nothing –

Returns 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`.

`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.

See also:

`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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns 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.

`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.

See also:

`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`.

Throws std::bad_alloc – (if one of the required allocations fails)

Returns 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.

`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.

See also:

`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`.

Throws nothing –

Returns 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.

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

`explicit 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.

`explicit exception(std::system_error const &e)`

Construct a `hpx::exception` from a `boost::system_error`.

`explicit 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 = throwmode::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 = throwmode::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() noexcept`

Destruct a `hpx::exception`

Throws nothing –

`error get_error() const noexcept`

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.

Throws nothing –

`error_code get_error_code(throwmode mode = throwmode::plain) const noexcept`

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 `throwmode::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
        // interruption now enabled
    }
```

At any point, the interruption state for the current thread can be queried by calling `hpx::this_thread::interruption_enabled()`.

hpx/errors/exception_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

Defines

`HPX_THROWMODE_UNSCOPED_ENUM_DEPRECATED_MSG`

namespace `hpx`

Enums

enum class `throwmode` : `std::uint8_t`
Encode error category for new `error_code`.
Values:

enumerator `plain`

enumerator `rethrow`

enumerator `lightweight`

Functions

inline constexpr bool `operator&(throwmode lhs, throwmode rhs)` noexcept

Variables

constexpr `throwmode plain = throwmode::plain`

constexpr `throwmode rethrow = throwmode::rethrow`

constexpr `throwmode lightweight = throwmode::lightweight`

constexpr `throwmode lightweight_rethrow = throwmode::lightweight_rethrow`

`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.

hpx/errors/exception_list.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

```
inline std::size_t size() const noexcept
```

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

Note: Complexity: Constant time.

```
inline exception_list_type::const_iterator begin() const noexcept
```

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

```
inline exception_list_type::const_iterator end() const noexcept
```

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

hpx/errors/throw_exception.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

namespace `hpx`

execution

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/execution/executors/adaptive_static_chunk_size.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **execution**

struct **adaptive_static_chunk_size**

#include <adaptive_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

inline constexpr **adaptive_static_chunk_size()** noexcept

Construct a `adaptive_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.

inline explicit constexpr **adaptive_static_chunk_size(*std::size_t* chunk_size)** noexcept

Construct a `adaptive_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 **parallel**

namespace **execution**

hpx/execution/executors/auto_chunk_size.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

inline constexpr **auto_chunk_size**(*std*::uint64_t num_iters_for_timing = 0) noexcept
 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.

inline explicit **auto_chunk_size**(*hpx::chrono*::steady_duration const &rel_time, *std*::uint64_t num_iters_for_timing = 0) noexcept
 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**

`hpx/execution/executors/dynamic_chunk_size.hpp`

See *Public API* for a list of names and headers that are part of the public HPX API.

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

inline explicit constexpr **dynamic_chunk_size**(*std*::size_t chunk_size = 1) noexcept
 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 **parallel**

namespace **execution**

hpx/execution/executors/execution.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **execution**

hpx/execution/executors/execution_information.hpp

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::has_pending_closures_t has_pending_closures

hpx::parallel::execution::get_pu_mask_t get_pu_mask

hpx::parallel::execution::set_scheduler_mode_t set_scheduler_mode

struct **get_pu_mask_t** : public *hpx::functional::detail::tag_fallback<get_pu_mask_t>*

#include <execution_information.hpp> Retrieve the bitmask describing the processing units the given thread is allowed to run on

All threads::executors invoke sched.get_pu_mask().

Note: If the executor does not support this operation, this call will always invoke hpx::threads::get_pu_mask()

Param exec [in] The executor object to use for querying the number of pending tasks.

Param topo [in] The topology object to use to extract the requested information.

Param thream_num [in] The sequence number of the thread to retrieve information for.

Private Functions

```
template<typename Executor>
inline decltype(auto) friend tag_fallback_invoke(get_pu_mask_t, Executor&&,  

                                                 threads::topology &topo, std::size_t  

                                                 thread_num)

template<typename Executor>
inline 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*

Param exec [in] The executor object to use to extract the requested information for.

Private Functions

```
template<typename Executor>
inline decltype(auto) friend tag_fallback_invoke(has_pending_closures_t, Executor&&)

template<typename Executor>
inline 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.

Param exec [in] The executor object to use.

Param mode [in] The new mode for the scheduler to pick up

Friends

```
template<typename Executor, typename Mode>
inline friend void tag_fallback_invoke(set_scheduler_mode_t, Executor&&, Mode const&)

template<typename Executor, typename Mode>
inline friend void tag_invoke(set_scheduler_mode_t, Executor &&exec, Mode const &mode)
```

hpx/execution/executors/execution_parameters.hpp

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 ...Params>
constexpr executor_parameters_join<Params...>::type join_executor_parameters(Params&&... params)
```

```
template<typename Param>
constexpr Param &&join_executor_parameters(Param &&param) noexcept
```

```
template<typename ...Params>
struct executor_parameters_join
```

Public Types

```
using type = detail::executor_parameters<std::decay_t<Params>...>
template<typename Param>
struct executor_parameters_join<Param>
```

Public Types

```
using type = Param
```

hpx/execution/executors/execution_parameters_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

template<>

```
struct is_scheduling_property<hpx::parallel::execution::with_processing_units_count_t> : public true_type
```

namespace **hpx**

 namespace **execution**

```

namespace experimental

template<> with_processing_units_count_t > : public true_type

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::with_processing_units_count_t with_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).

Param params [in] The executor parameters object to use for determining the chunk size for the given number of tasks *num_tasks*.
Param exec [in] The executor object which will be used for scheduling of the loop iterations.
Param f [in] The function which will be optionally scheduled using the given executor.
Param cores [in] The number of cores the number of chunks should be determined for.
Param num_tasks [in] The number of tasks the chunk size should be determined for

Private Functions

```
template<typename Parameters, typename Executor, typename F>
inline 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::tagFallback<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.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
inline 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.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
inline 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.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
inline decltype(auto) friend tagFallbackInvoke(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.
```

Param params [in] The executor parameters object to use for determining the number of chunks for the given number of *cores*.

Param exec [in] The executor object which will be used for scheduling of the loop iterations.

Param cores [in] The number of cores the number of chunks should be determined for.

Param num_tasks [in] The number of tasks the chunk size should be determined for

Private Functions

```
template<typename Parameters, typename Executor>
inline decltype(auto) friend tagFallbackInvoke(maximal_number_of_chunks_t,
Parameters &&params, Executor
&&exec, std::size_t cores, std::size_t
num_tasks)
```

```
struct processing_units_count_t : public
hpx::functional::detail::tagFallback<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.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```
template<typename Parameters, typename Executor>
inline decltype(auto) friend tagFallbackInvoke(processing_units_count_t, Parameters
&&params, Executor &&exec)
```

```
struct reset_thread_distribution_t : public
hpx::functional::detail::tagFallback<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.

Param params [in] The executor parameters object to use for resetting the thread distribution scheme.
Param exec [in] The executor object to use.

Private Functions

```
template<typename Parameters, typename Executor>
inline decltype(auto) friend tag_fallback_invoke(reset_thread_distribution_t, Parameters
&&params, Executor &&exec)
```

```
struct with_processing_units_count_t : public
hpx::functional::detail::tag_fallback<with_processing_units_count_t>
#include <execution_parameters_fwd.hpp> Generate a policy that supports setting the number of
cores for execution.
```

[hpx/execution/executors/guided_chunk_size.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

```
inline explicit constexpr guided_chunk_size(std::size_t min_chunk_size = 1) noexcept
```

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 **parallel**

namespace **execution**

hpx/execution/executors/num_cores.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **execution**

struct num_cores

#include <num_cores.hpp> Control number of cores in executors which need a functionality for setting the number of cores to be used by an algorithm directly

Public Functions

inline explicit constexpr num_cores(*std::size_t* cores = 1) noexcept

 Construct a **num_cores** executor parameters object

Note: make sure the minimal number of cores is 1

 namespace **parallel**

 namespace **execution**

hpx/execution/executors/persistent_auto_chunk_size.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

```
inline constexpr persistent_auto_chunk_size(std::uint64_t num_iters_for_timing = 0)  
noexcept
```

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.

```
inline explicit persistent_auto_chunk_size(hpx::chrono::steady_duration const &time_cs,  
                                         std::uint64_t num_iters_for_timing = 0) noexcept
```

Construct an `persistent_auto_chunk_size` executor parameters object

Parameters `time_cs` – The execution time for each chunk.

```
inline 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) noexcept
```

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 `parallel`

namespace `execution`

[hpx/execution/executors/polymorphic_executor.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace `hpx`

namespace `parallel`

namespace `execution`

```
template<typename Sig>
```

```
class polymorphic_executor
```

```
template<typename R, typename ...Ts>
```

```
class polymorphic_executor<R(Ts...)> : private  
    hpx::parallel::execution::detail::polymorphic_executor_base
```

Public Types

```
template<typename>
using future_type = hpx::future<R>
```

Public Functions

```
inline constexpr polymorphic_executor() noexcept
inline polymorphic_executor(polymorphic_executor const &other)
inline polymorphic_executor(polymorphic_executor &&other) noexcept
inline polymorphic_executor &operator=(polymorphic_executor const &other)
inline polymorphic_executor &operator=(polymorphic_executor &&other) noexcept
template<typename Exec, typename PE = std::decay_t<Exec>, typename Enable =
std::enable_if_t<!std::is_same_v<PE, polymorphic_executor>>>
inline polymorphic_executor(Exec &&exec)

template<typename Exec, typename PE = std::decay_t<Exec>, typename Enable =
std::enable_if_t<!std::is_same_v<PE, polymorphic_executor>>>
inline polymorphic_executor &operator=(Exec &&exec)

inline void reset() noexcept
```

Private Types

```
using base_type = detail::polymorphic_executor_base

using vtable = detail::polymorphic_executor_vtable<R(Ts...)>
```

Private Functions

```
inline void assign(std::nullptr_t) noexcept
template<typename Exec>
inline void assign(Exec &&exec)
```

Private Static Functions

```
static inline constexpr vtable const *get_empty_vtable() noexcept
template<typename T>
static inline constexpr vtable const *get_vtable() noexcept
```

Friends

```
template<typename F>
inline friend void tag_invoke(hpx::parallel::execution::post_t, polymorphic_executor const
&exec, F &&f, Ts... ts)
```

```
template<typename F>
inline friend R tag_invoke(hpx::parallel::execution::sync_execute_t, polymorphic_executor
const &exec, F &&f, Ts... ts)
```

```
template<typename F>
inline friend hpx::future<R> tag_invoke(hpx::parallel::execution::async_execute_t,
polymorphic_executor const &exec, F &&f, Ts... ts)
```

```
template<typename F, typename Future>
inline friend hpx::future<R> tag_invoke(hpx::parallel::execution::then_execute_t,
polymorphic_executor const &exec, F &&f, Future
&&predecessor, Ts&&... ts)
```

```
template<typename F, typename Shape>
inline friend std::vector<R> tag_invoke(hpx::parallel::execution::bulk_sync_execute_t,
polymorphic_executor const &exec, F &&f, Shape
const &s, Ts&&... ts)
```

```
template<typename F, typename Shape>
inline friend std::vector<hpx::future<R>> tag_invoke(hpx::parallel::execution::bulk_async_execute_t,
polymorphic_executor const &exec, F
&&f, Shape const &s, Ts&&... ts)
```

```
template<typename F, typename Shape>
inline friend hpx::future<std::vector<R>> tag_invoke(hpx::parallel::execution::bulk_then_execute_t,
polymorphic_executor const &exec, F
&&f, Shape const &s,
hpx::shared_future<void> const
&predecessor, Ts&&... ts)
```

[*hpx/execution/executors/rebind_executor.hpp*](#)

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

```
constexpr create_rebound_policy_t create_rebound_policy = {}
```

```
struct create_rebound_policy_t
```

Public Functions

```
template<typename ExPolicy, typename Executor, typename Parameters>
inline 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

```
using type = typename policy_type::template rebind<executor_type, parameters_type>::type
```

The type of the rebound execution policy.

hpx/execution/executors/static_chunk_size.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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

```
inline constexpr static_chunk_size() noexcept  
    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.

```
inline explicit constexpr static_chunk_size(std::size_t chunk_size) noexcept  
    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 parallel
```

```
    namespace execution
```

`hpx/execution/traits/is_execution_policy.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
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<T>>  
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy makes algorithms asynchronous
```

- i. 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.

ii. 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>`.

iii. 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<T>>
```

```
#include <is_execution_policy.hpp>
```

i. 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.

ii. 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>`.

iii. 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<T>>
```

```
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy enables parallelization
```

i. 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.

ii. 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>`.

iii. 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<T>>
```

```
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy does not enable parallelization
```

i. 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.

ii. 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>`.

iii. The behavior of a program that adds specializations for `is_sequenced_execution_policy` is undefined.

execution_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/execution_base/execution.hpp

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

struct **parallel_execution_tag**

#include <execution.hpp> Function invocations executed by a group of parallel execution agents execute in unordered fashion. Any such invocations executing in the same thread are indeterminately sequenced with respect to each other.

Note: `parallel_execution_tag` is weaker than `sequenced_execution_tag`.

struct **sequenced_execution_tag**

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

struct **unsequenced_execution_tag**

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

Note: `unsequenced_execution_tag` is weaker than `parallel_execution_tag`.

namespace **parallel**

namespace **execution**

Variables

hpx::parallel::execution::sync_execute_t sync_execute

hpx::parallel::execution::async_execute_t async_execute

hpx::parallel::execution::then_execute_t then_execute

`hpx::parallel::execution::post_t post`

`hpx::parallel::execution::bulk_sync_execute_t bulk_sync_execute`

`hpx::parallel::execution::bulk_async_execute_t bulk_async_execute`

`hpx::parallel::execution::bulk_then_execute_t bulk_then_execute`

```
struct async_execute_t : public hpx::functional::detail::tag_fallback<async_execute_t>
    #include <execution.hpp> Customization point for asynchronous execution agent creation.
```

This asynchronously creates a single function invocation `f()` using the associated executor.

Note: Executors have to implement only `async_execute()`. All other functions will be emulated by this or other customization points in terms of this single basic primitive. However, some executors will naturally specialize all operations for maximum efficiency.

Note: This is valid for one way executors (calls `make_ready_future(exec.sync_execute(f, ts...))` if it exists) and for two way executors (calls `exec.async_execute(f, ts...)` if it exists).

Param exec [in] The executor object to use for scheduling of the function `f`.
Param f [in] The function which will be scheduled using the given executor.
Param ts [in] Additional arguments to use to invoke `f`.
Return `f(ts...)`'s result through a future

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(async_execute_t, Executor &&exec, F
    &&f, Ts&&... ts)
```

```
struct bulk_async_execute_t : public
hpx::functional::detail::tag_fallback<bulk_async_execute_t>
    #include <execution.hpp> Bulk form of asynchronous execution agent creation.
```

This asynchronously creates a group of function invocations `f(i)` whose ordering is given by the `execution_category` associated with the executor.

Here `i` takes on all values in the index space implied by shape. All exceptions thrown by invocations of `f(i)` are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This is deliberately different from the `bulk_async_execute` customization points specified in P0443. The `bulk_async_execute` customization point defined here is more generic and is used as the workhorse for implementing the specified APIs.

Note: This calls `exec.bulk_async_execute(f, shape, ts...)` if it exists; otherwise it executes `async_execute(f, shape, ts...)` as often as needed.

Param exec [in] The executor object to use for scheduling of the function *f*.

Param f [in] The function which will be scheduled using the given executor.

Param shape [in] The shape objects which defines the iteration boundaries for the arguments to be passed to *f*.

Param ts [in] Additional arguments to use to invoke *f*.

Return The return type of `executor_type::bulk_async_execute` if defined by `executor_type`.

Otherwise a vector of futures holding the returned values of each invocation of *f*.

Private Functions

```
template<typename Executor, typename F, typename Shape, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(bulk_async_execute_t, Executor &&exec,
                                              F &&f, Shape const &shape, Ts&&... ts)
```

```
template<typename Executor, typename F, typename Shape, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(bulk_async_execute_t, Executor &&exec,
                                              F &&f, Shape const &shape, Ts&&... ts)
```

```
struct bulk_sync_execute_t : public hpx::functional::detail::tag_fallback<bulk_sync_execute_t>
#include <execution.hpp> Bulk form of synchronous execution agent creation.
```

This synchronously creates a group of function invocations *f(i)* whose ordering is given by the `execution_category` associated with the executor. The function synchronizes the execution of all scheduled functions with the caller.

Here *i* takes on all values in the index space implied by `shape`. All exceptions thrown by invocations of *f(i)* are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This is deliberately different from the `bulk_sync_execute` customization points specified in P0443. The `bulk_sync_execute` customization point defined here is more generic and is used as the workhorse for implementing the specified APIs.

Note: This calls `exec.bulk_sync_execute(f, shape, ts...)` if it exists; otherwise it executes `sync_execute(f, shape, ts...)` as often as needed.

Param exec [in] The executor object to use for scheduling of the function *f*.

Param f [in] The function which will be scheduled using the given executor.

Param shape [in] The shape objects which defines the iteration boundaries for the arguments to be passed to *f*.

Param ts [in] Additional arguments to use to invoke *f*.

Return The return type of `executor_type::bulk_sync_execute` if defined by `executor_type`.

Otherwise a vector holding the returned values of each invocation of *f* except when *f* returns void, which case void is returned.

Private Functions

```
template<typename Executor, typename F, typename Shape, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(bulk_sync_execute_t, Executor &&exec,
                                                F &&f, Shape const &shape, Ts&&... ts)
```

```
template<typename Executor, typename F, typename Shape, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(bulk_sync_execute_t, Executor &&exec,
                                                F &&f, Shape const &shape, Ts&&... ts)
```

```
struct bulk_then_execute_t : public hpx::functional::detail::tag_fallback<bulk_then_execute_t>
#include <execution.hpp> Bulk form of execution agent creation depending on a given future.
```

This creates a group of function invocations $f(i)$ whose ordering is given by the execution_category associated with the executor.

Here i takes on all values in the index space implied by shape. All exceptions thrown by invocations of $f(i)$ are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This is deliberately different from the then_sync_execute customization points specified in P0443. The bulk_then_execute customization point defined here is more generic and is used as the workhorse for implementing the specified APIs.

Note: This calls exec.bulk_then_execute(f, shape, pred, ts...) if it exists; otherwise it executes sync_execute(f, shape, pred.share(), ts...) (if this executor is also an OneWayExecutor), or async_execute(f, shape, pred.share(), ts...) (if this executor is also a TwoWayExecutor) - as often as needed.

Param exec [in] The executor object to use for scheduling of the function f .

Param f [in] The function which will be scheduled using the given executor.

Param shape [in] The shape objects which defines the iteration boundaries for the arguments to be passed to f .

Param predecessor [in] The future object the execution of the given function depends on.

Param ts [in] Additional arguments to use to invoke f .

Return The return type of $executor_type::bulk_then_execute$ if defined by $executor_type$. Otherwise a vector holding the returned values of each invocation of f .

Private Functions

```
template<typename Executor, typename F, typename Shape, typename Future, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(bulk_then_execute_t, Executor &&exec,
                                                F &&f, Shape const &shape, Future &&predecessor, Ts&&... ts)
```

```
template<typename Executor, typename F, typename Shape, typename Future, typename ...Ts>
```

```
inline decltype(auto) friend tag_fallback_invoke(bulk_then_execute_t, Executor &&exec,  
    F &&f, Shape const &shape, Future  
    &&predecessor, Ts&&... ts)
```

```
struct post_t : public hpx::functional::detail::tag_fallback<post_t>
```

```
#include <execution.hpp> Customization point for asynchronous fire & forget execution agent  
creation.
```

This asynchronously (fire & forget) creates a single function invocation *f()* using the associated executor.

Note: This is valid for two way executors (calls *exec.post(f, ts...)*, if available, otherwise it calls *exec.async_execute(f, ts...)* while discarding the returned future), and for non-blocking two way executors (calls *exec.post(f, ts...)* if it exists).

Param exec [in] The executor object to use for scheduling of the function *f*.

Param f [in] The function which will be scheduled using the given executor.

Param ts [in] Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>  
inline decltype(auto) friend tag_fallback_invoke(post_t, Executor &&exec, F &&f,  
    Ts&&... ts)
```

```
struct sync_execute_t : public hpx::functional::detail::tag_fallback<sync_execute_t>
```

```
#include <execution.hpp> Customization point for synchronous execution agent creation.
```

This synchronously creates a single function invocation *f()* using the associated executor. The execution of the supplied function synchronizes with the caller

Note: It will call *tag_invoke(sync_execute_t, exec, f, ts...)* if it exists. For two-way executors it will invoke *asynch_execute_t* and wait for the task's completion before returning.

Param exec [in] The executor object to use for scheduling of the function *f*.

Param f [in] The function which will be scheduled using the given executor.

Param ts [in] Additional arguments to use to invoke *f*.

Return *f(ts...)*'s result

Private Functions

```
template<typename Executor, typename F, typename ...Ts>  
inline decltype(auto) friend tag_fallback_invoke(sync_execute_t, Executor &&exec, F  
    &&f, Ts&&... ts)
```

```
struct then_execute_t : public hpx::functional::detail::tag_fallback<then_execute_t>
```

```
#include <execution.hpp> Customization point for execution agent creation depending on a given  
future.
```

This creates a single function invocation `f()` using the associated executor after the given future object has become ready.

Note: This is valid for two way executors (calls `exec.then_execute(f, predecessor, ts...)` if it exists) and for one way executors (calls `predecessor.then(bind(f, ts...))`).

Param exec [in] The executor object to use for scheduling of the function `f`.

Param f [in] The function which will be scheduled using the given executor.

Param predecessor [in] The future object the execution of the given function depends on.

Param ts [in] Additional arguments to use to invoke `f`.

Return `f(ts...)`'s result through a future

Private Functions

```
template<typename Executor, typename F, typename Future, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(then_execute_t, Executor &&exec, F
&&f, Future &&predecessor, Ts&&... ts)
```

hpx/execution_base/receiver.hpp

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 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_stopped(R &&r)
```

`set_stopped` is a customization point object. The expression `hpx::execution::set_stopped(r)` is equivalent to:

- `r.set_stopped()`, 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, `set_stopped(r)`, if that expression is valid, with overload resolution performed in a context that include the declaration `void set_stopped();`
- 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_stopped(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_stopped_t set_stopped
```

```
template<typename T, typename E = std::exception_ptr>
```

```
constexpr bool is_receiver_v = is_receiver<T, E>::value
```

```
template<typename T, typename CS>
```

```
constexpr bool is_receiver_of_v = is_receiver_of<T, CS>::value
```

```
template<typename T, typename CS>
```

```
constexpr bool is_nothrow_receiver_of_v = is_nothrow_receiver_of<T, CS>::value
```

```
template<typename T, typename CS>
```

```
struct is_nothrow_receiver_of : public
```

```
hpx::execution::experimental::detail::is_nothrow_receiver_of_impl<is_receiver_v<T> &&
is_receiver_of_v<T, CS>, T, CS>
```

```
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_stopped`
- `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 also:

`hpx::execution::experimental::is_receiver_of`

template<typename T, typename CS>

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`

The `receiver_of` concept takes a receiver and an instance of the `completion_signatures`<> class template. The `receiver_of` concept, rather than accepting a receiver and some value types, is changed to take a receiver and an instance of the `completion_signatures`<> class template. A sender uses `completion_signatures`<> to describe the signals with which it completes. The `receiver_of` concept ensures that a particular receiver is capable of receiving those signals.

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_stopped`

See also:

`hpx::execution::traits::is_receiver`

struct `set_error_t` : public `hpx::functional::tag_noexcept<set_error_t>`

struct `set_stopped_t` : public `hpx::functional::tag_noexcept<set_stopped_t>`

struct `set_value_t` : public `hpx::functional::tag<set_value_t>`

`hpx/execution_base/traits/is_executor_parameters.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

template<typename Executor>

struct `hpx::parallel::execution::extract_executor_parameters<Executor, std::void_t<typename Executor::executor_parameters_type>>`

Public Types

```
using type = typename Executor::executor_parameters_type

template<typename Parameters>

struct extract_has_variable_chunk_size<Parameters, std::void_t<typename Parameters::has_variable_chunk_size>> : public true_type

namespace hpx
```

```
    namespace parallel
```

```
        namespace execution
```

Typedefs

```
template<typename Executorextract_executor_parameters_t = typename
extract_executor_parameters<Executor>::type

template<typename T>

using is_executor_parameters_t = typename is_executor_parameters<T>::type
```

Variables

```
template<typename Parameters>

constexpr bool extract_has_variable_chunk_size_v =
extract_has_variable_chunk_size<Parameters>::value

template<typename Parameters>

constexpr bool extract_invokes_testing_function_v =
extract_invokes_testing_function<Parameters>::value

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

```
using type = sequential_executor_parameters

template<typename Executor> executor_parameters_type > >
```

Public Types

```
using type = typename Executor::executor_parameters_type

template<typename Parameters, typename Enable = void>
struct extract_has_variable_chunk_size : public false_type

template<typename Parameters> has_variable_chunk_size > > : public true_type
template<typename Parameters, typename Enable = void>
struct extract_invokes_testing_function : public false_type
template<typename T>
struct is_executor_parameters : public detail::is_executor_parameters<std::decay_t<T>>

struct sequential_executor_parameters

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
template<typename Parameters, typename Enable>
struct is_executor_parameters
```

executors

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/executors/annotating_executor.hpp

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 Tag, typename BaseExecutor, typename Property>
auto tag_invoke(Tag tag, annotating_executor<BaseExecutor> const &exec, Property &&prop)
    -> decltype(
        cltype(annotating_executor<BaseExecutor>(std::declval<Tag>())(std::declval<BaseExecutor>(),
        std::declval<Property>())))
    -> de-
```

```
template<typename Tag, typename BaseExecutor>
auto tag_invoke(Tag tag, annotating_executor<BaseExecutor> const &exec) ->
    decltype(std::declval<Tag>()(std::declval<BaseExecutor>()))
```

```
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> &&
    !std::is_same_v<std::decay_t<Executor>, annotating_executor>>>
inline explicit 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>>>>
inline explicit annotating_executor(Executor &&exec, std::string annotation)
```

```
namespace parallel
```

```
    namespace execution
```

hpx/executors/current_executor.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
    namespace parallel
```

```
        namespace execution
```

Typedefs

```
typedef hpx::execution::parallel_executor instead
```

```
namespace this_thread
```

Functions

```
hpx::execution::parallel_executor get_executor(error_code &ec = throws)
```

Returns a reference to the executor which was used to create the current thread.

Throws 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

```
hpx::execution::parallel_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.

Throws 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*.

hpx/executors/exception_list.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

hpx/executors/execution_policy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **execution**

Typedefs

using **sequenced_task_policy** = detail::sequenced_task_policy_shim<*sequenced_executor*>

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.

using **sequenced_policy** = detail::sequenced_policy_shim<*sequenced_executor*>

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.

using **parallel_task_policy** = detail::parallel_task_policy_shim<*parallel_executor*>

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.

using **parallel_policy** = detail::parallel_policy_shim<*parallel_executor*>

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.

using **parallel_unsequenced_task_policy** =
detail::parallel_unsequenced_task_policy_shim<*parallel_executor*>

The class `parallel_unsequenced_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 and vectorized.

```
using parallel_unsequenced_policy =  
    detail::parallel_unsequenced_policy_shim<parallel_executor>
```

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.

```
using unsequenced_task_policy = detail::unsequenced_task_policy_shim<sequenced_executor>
```

The class `unsequenced_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 vectorized.

```
using unsequenced_policy = detail::unsequenced_policy_shim<sequenced_executor>
```

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.

Variables

```
constexpr task_policy_tag task = {}
```

```
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 non_task_policy_tag : public hpx::execution::experimental::to_non_task_t
```

```
struct task_policy_tag : public hpx::execution::experimental::to_task_t
```

```
namespace experimental
```

```
template<>
```

```
struct is_execution_policy_mapping<non_task_policy_tag> : public true_type
```

```
template<>
```

```
struct is_execution_policy_mapping<task_policy_tag> : public true_type
```

hpx/executors/execution_policy_annotation.hpp

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 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)
```

hpx/executors/execution_policy_mappings.hpp

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

```
template<typename Tag>
constexpr bool is_execution_policy_mapping_v = is_execution_policy_mapping<Tag>::value

hpx::execution::experimental::to_non_par_t to_non_par

hpx::execution::experimental::to_par_t to_par
```

```

hpx::execution::experimental::to_non_task_t to_non_task

hpx::execution::experimental::to_task_t to_task

hpx::execution::experimental::to_non_unseq_t to_non_unseq

hpx::execution::experimental::to_unseq_t to_unseq

template<typename Tag>
struct is_execution_policy_mapping : public false_type
template<>
struct is_execution_policy_mapping<to_non_par_t> : public true_type
template<>
struct is_execution_policy_mapping<to_non_task_t> : public true_type
template<>
struct is_execution_policy_mapping<to_non_unseq_t> : public true_type
template<>
struct is_execution_policy_mapping<to_par_t> : public true_type
template<>
struct is_execution_policy_mapping<to_task_t> : public true_type
template<>
struct is_execution_policy_mapping<to_unseq_t> : public true_type

struct to_non_par_t : public hpx::functional::detail::tag_fallback<to_non_par_t>

```

Private Functions

```

template<typename ExPolicytagFallbackInvoke(to_non_par_t, ExPolicy
&&policy) noexcept

```

```

struct to_non_task_t : public hpx::functional::detail::tag_fallback<to_non_task_t>
Subclassed by hpx::execution::non_task_policy_tag

```

Private Functions

```
template<typename ExPolicy>
inline constexpr decltype(auto) friend tag_fallback_invoke(to_non_task_t, ExPolicy
&&policy) noexcept
```

```
struct to_non_unseq_t : public hpx::functional::detail::tag_fallback<to_non_unseq_t>
```

Private Functions

```
template<typename ExPolicy>
inline constexpr decltype(auto) friend tag_fallback_invoke(to_non_unseq_t, ExPolicy
&&policy) noexcept
```

```
struct to_par_t : public hpx::functional::detail::tag_fallback<to_par_t>
```

Private Functions

```
template<typename ExPolicy>
inline constexpr decltype(auto) friend tag_fallback_invoke(to_par_t, ExPolicy &&policy)
noexcept
```

```
struct to_task_t : public hpx::functional::detail::tag_fallback<to_task_t>
```

Subclassed by *hpx::execution::task_policy_tag*

Private Functions

```
template<typename ExPolicy>
inline constexpr decltype(auto) friend tag_fallback_invoke(to_task_t, ExPolicy &&policy)
noexcept
```

```
struct to_unseq_t : public hpx::functional::detail::tag_fallback<to_unseq_t>
```

Private Functions

```
template<typename ExPolicy>
inline constexpr decltype(auto) friend tag_fallback_invoke(to_unseq_t, ExPolicy
&&policy) noexcept
```

hpx/executors/execution_policy_parameters.hpp

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 ExPolicy>
constexpr decltype(auto) tag_invoke(with_processing_units_count_t, ExPolicy &&policy,
                                         std::size_t num_cores)

template<typename ExPolicy, typename Params>
constexpr decltype(auto) tag_invoke(with_processing_units_count_t, ExPolicy &&policy, Params
                                         &&params)

template<typename ParametersProperty, typename ExPolicy, typename Params>
constexpr decltype(auto) tag_fallback_invoke(ParametersProperty, ExPolicy &&policy,
                                              Params &&params)

template<typename ParametersProperty, typename ExPolicy, typename ...Ts>
constexpr auto tag_fallback_invoke(ParametersProperty prop, ExPolicy &&policy, Ts&&... ts)
    -> decltype(std::declval<ParametersProperty>()(std::declval<typename
        std::decay_t<ExPolicy>::executor_type>(),
        std::declval<Ts>(...)))
```

hpx/executors/explicit_scheduler_executor.hpp

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 BaseScheduler>
explicit explicit_scheduler_executor(BaseScheduler &&sched) -> ex-
plicit_scheduler_executor<std::decay_t<BaseScheduler>>

template<typename Tag, typename BaseScheduler, typename Propertytag_invoke(Tag tag, explicit_scheduler_executor<BaseScheduler> const &exec, Property
&&prop) -> de-
cltype(explicit_scheduler_executor<BaseScheduler>(std::declval<Tag>()(std::declval<BaseScheduler>(),
std::declval<Property>())))

template<typename Tag, typename BaseScheduler, typename ...Ts>
auto tag_invoke(Tag tag, explicit_scheduler_executor<BaseScheduler> const &exec, Ts&&... ts)
-> decltype(std::declval<Tag>()(std::declval<BaseScheduler>(),
std::declval<Ts>(...)))

template<typename BaseSchedulerexplicit_scheduler_executor

namespace parallel

namespace execution
```

[hpx/executors/fork_join_executor.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

[hpx/executors/parallel_executor.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **execution**

Typedefs

```
using parallel_executor = parallel_policy_executor<hpx::launch>
```

Functions

```
template<typename Tag, typename Policy, typename Property>
auto tag_invoke(Tag tag, parallel_policy_executor<Policy> const &exec, Property &&prop) -> decltype(std::declval<parallel_policy_executor<Policy>>().policy(std::declval<Tag>()(std::declval<Policy>(&prop)), std::declval<Property>()), parallel_policy_executor<Policy>())

template<typename Tag, typename Policy>
auto tag_invoke(Tag tag, parallel_policy_executor<Policy> const &exec) ->
    decltype(std::declval<Tag>()(std::declval<Policy>()))

template<typename Policy>
struct parallel_policy_executor
{
    #include <parallel_executor.hpp> A parallel_executor creates groups of parallel execution agents which execute in threads implicitly created by the executor. This executor prefers continuing with the creating thread first before executing newly created threads.
}
```

This executor conforms to the concepts of a TwoWayExecutor, and a BulkTwoWayExecutor

Public Types

```
using execution_category = std::conditional_t<std::is_same_v<Policy, launch::sync_policy>, sequenced_execution_tag, parallel_execution_tag>
```

Associate the parallel_execution_tag executor tag type as a default with this executor, except if the given launch policy is synch.

```
using executor_parameters_type = static_chunk_size
```

Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

```
inline explicit constexpr parallel_policy_executor(threads::thread_priority priority,
                                                 threads::thread_stacksize stacksize =
                                                 threads::thread_stacksize::default_,
                                                 threads::thread_schedule_hint
                                                 schedulehint = {}, Policy l = parallel::execution::detail::get_default_policy<Policy>::call(),
                                                 std::size_t hierarchical_threshold =
                                                 hierarchical_threshold_default_)
```

Create a new parallel executor.

```
inline explicit constexpr parallel_policy_executor(threads::thread_stacksize stacksize,
                                                 threads::thread_schedule_hint
                                                 schedulehint = {}, Policy l = parallel::execution::detail::get_default_policy<Policy>::call())
```

```
inline explicit constexpr parallel_policy_executor(threads::thread_schedule_hint
                                                 schedulehint, Policy l = parallel::execution::detail::get_default_policy<Policy>::call())
```

```
inline explicit constexpr parallel_policy_executor(Policy l = parallel::execution::detail::get_default_policy<Policy>::call())  
  
inline explicit constexpr parallel_policy_executor(threads::thread_pool_base *pool,  
                                              threads::thread_priority priority =  
                                              threads::thread_priority::default_,  
                                              threads::thread_stacksize stacksize =  
                                              threads::thread_stacksize::default_,  
                                              threads::thread_schedule_hint  
                                              schedulehint = {}, Policy l = parallel::execution::detail::get_default_policy<Policy>::call(),  
                                              std::size_t hierarchical_threshold =  
                                              hierarchical_threshold_default_)
```

template<typename **Parameters**>
inline *std::size_t* **processing_units_count**(*Parameters*&&) const

Friends

```
inline friend constexpr friend parallel_policy_executor tag_invoke (hpx::parallel::execution::parallel_policy_executor const &exec, std::size_t num_cores)  
  
inline friend constexpr friend std::size_t tag_invoke (hpx::parallel::execution::processing::parallel_policy_executor const &exec)  
  
inline friend auto tag_invoke(hpx::execution::experimental::get_processing_units_mask_t,  
                           parallel_policy_executor const &exec)  
  
inline friend auto tag_invoke(hpx::execution::experimental::get_cores_mask_t,  
                           parallel_policy_executor const &exec)
```

namespace **parallel**

namespace **execution**

[hpx/executors/parallel_executor_aggregated.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **parallel**

namespace **execution**

`hpx/executors/restricted_thread_pool_executor.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **execution**

Typedefs

```
using restricted_thread_pool_executor = restricted_policy_executor<hpx::launch>
template<typename Policy>
class restricted_policy_executor
```

Public Types

```
using execution_category = typename embedded_executor::execution_category
Associate the parallel_execution_tag executor tag type as a default with this executor.
```

```
using executor_parameters_type = typename
embedded_executor::executor_parameters_type
Associate the static_chunk_size executor parameters type as a default with this executor.
```

Public Functions

```
inline explicit restricted_policy_executor(std::size_t first_thread = 0, std::size_t
                                         num_threads = 1, threads::thread_priority
                                         priority = threads::thread_priority::default_,
                                         threads::thread_stacksize stacksize =
                                         threads::thread_stacksize::default_,
                                         threads::thread_schedule_hint schedulehint =
                                         {}, std::size_t hierarchical_threshold =
                                         hierarchical_threshold_default_)
```

Create a new parallel executor.

```
inline restricted_policy_executor(restricted_policy_executor const &other)
```

Private Types

```
using embedded_executor = hpx::execution::parallel_policy_executor<Policy>
```

Private Members

```
const std::uint16_t first_thread_
```

```
mutable std::atomic<std::size_t> os_thread_
```

```
embedded_executor exec_
```

Private Static Attributes

```
static constexpr std::size_t hierarchical_threshold_default_ = 6
```

hpx/executors/scheduler_executor.hpp

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 BaseScheduler>
explicit scheduler_executor(BaseScheduler &&sched) ->
    scheduler_executor<std::decay_t<BaseScheduler>>

template<typename Tag, typename BaseScheduler, typename Property>
auto tag_invoke(Tag tag, scheduler_executor<BaseScheduler> const &exec, Property &&prop)
    -> decltype(scheduler_executor<BaseScheduler>(std::declval<Tag>()(std::declval<BaseScheduler>0,
        std::declval<Property>())))

template<typename Tag, typename BaseScheduler>
auto tag_invoke(Tag tag, scheduler_executor<BaseScheduler> const &exec) ->
    decltype(std::declval<Tag>()(std::declval<BaseScheduler>0))

template<typename BaseScheduler>
struct scheduler_executor
```

```
namespace parallel
```

```
    namespace execution
```

hpx/executors/sequenced_executor.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
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 parallel
```

```
    namespace execution
```

hpx/executors/service_executors.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
    namespace parallel
```

```
        namespace execution
```

hpx/executors/std_execution_policy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/executors/thread_pool_scheduler.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
    namespace execution
```

```
        namespace experimental
```

TypeDefs

```
using thread_pool_scheduler = thread_pool_policy_scheduler<hpx::launch>
```

Functions

```
template<typename Tag, typename Policy, typename Propertytag_invoke(Tag tag, thread_pool_policy_scheduler<Policy> const &scheduler, Property
&&prop) -> de-
cltype(std::declval<thread_pool_policy_scheduler<Policy>>().policy(std::declval<Tag>()(std::decl-
val<Property>()))), thread_pool_policy_scheduler<Policy>()
```



```
template<typename Tag, typename Policy>
auto tag_invoke(Tag tag, thread_pool_policy_scheduler<Policy> const &scheduler) ->
dectype(std::declval<Tag>()(std::declval<Policy>()))
```



```
template<typename Policy>
struct thread_pool_policy_scheduler
```

Public Types

```
using execution_category = std::conditional_t<std::is_same_v<Policy,
launch::sync_policy>, sequenced_execution_tag, parallel_execution_tag>
```

Public Functions

```
inline constexpr thread_pool_policy_scheduler(Policy l = experimen-
tal::detail::get_default_scheduler_policy<Policy>::call())
```



```
inline explicit thread_pool_policy_scheduler(hpx::threads::thread_pool_base *pool,
Policy l = experimen-
tal::detail::get_default_scheduler_policy<Policy>::call())
```

[hpx/executors/datapar/execution_policy.hpp](#)

See *Public API* for a list of names and headers that are part of the public *HPX* API.

[hpx/executors/datapar/execution_policy_mappings.hpp](#)

See *Public API* for a list of names and headers that are part of the public *HPX* API.

filesystem

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/modules/filesystem.hpp

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

```
inline path initial_path()
inline std::string basename(path const &p)
inline path canonical(path const &p, path const &base)
inline path canonical(path const &p, path const &base, std::error_code &ec)
```

namespace **filesystem**

functional

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/functional/bind.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Top level namespace.

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(F
&&f,
Ts&&...
vs)
```

The function template *bind* generates a forwarding call wrapper for *f*. Calling this wrapper is equivalent to invoking *f* with some of its arguments bound to *vs*.

Parameters

- **f** – Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments
- **vs** – list of arguments to bind, with the unbound arguments replaced by the placeholders `_1, _2, _3...` of namespace `hpx::placeholders`

Returns A function object of unspecified type *T*, for which

```
hpx::is_bind_expression<T>::value == true.
```

namespace `placeholders`

The `hpx::placeholders` namespace contains the placeholder objects `[_1, ..., _N]` where *N* is an implementation defined maximum number. When used as an argument in a `hpx::bind` expression, the placeholder objects are stored in the generated function object, and when that function object is invoked with unbound arguments, each placeholder `_N` is replaced by the corresponding *N*th unbound argument. The types of the placeholder objects are `DefaultConstructible` and `CopyConstructible`, their default copy/move constructors do not throw exceptions, and for any placeholder `_N`, the type `hpx::is_placeholder<decltype(_N)>` is defined, where `hpx::is_placeholder<decltype(_N)>` is derived from `std::integral_constant<int, N>`.

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 `serialization`

Functions

```
template<typename Archive, typename F, typename ...Ts>
void serialize(Archive &ar, ::hpx::detail::bound<F, Ts...> &bound, unsigned int const version = 0)

template<typename Archive, std::size_t I>
constexpr void serialize(Archive&, ::hpx::detail::placeholder<I>&, unsigned int const = 0) noexcept

namespace traits
```

```
namespace util
```

Functions

```
template<typename F, typename... Ts> HPX_DEPRECATED_V (1, 8,
"hpvx::util::bind is deprecated,
use hpx::bind instead") const expr decltype(auto) bind(F &&f
```

Variables

```
Ts && ts {return hpx::bind(HPX_FORWARD(F, f), HPX_FORWARD(Ts, ts)...)
```

```
namespace placeholders
```

Functions

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_1 is deprecated,
use " "hpx::placeholders::_1 instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_2 is deprecated,
use " "hpx::placeholders::_2 instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_3 is deprecated,
use " "hpx::placeholders::_3 instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_4 is deprecated,
use " "hpx::placeholders::_4 instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_5 is deprecated,
use " "hpx::placeholders::_5 instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_6 is deprecated,
use " "hpx::placeholders::_6 instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_7 is deprecated,  
use \"hpx::placeholders::_7 instead\") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_8 is deprecated,  
use \"hpx::placeholders::_8 instead\") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_9 is deprecated,  
use \"hpx::placeholders::_9 instead\") inline const expr hpx
```

hpx/functional/bind_front.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Top level namespace.

Functions

```
template<typename F, typename ...Ts>  
constexpr detail::bound_front<std::decay_t<F>, util::make_index_pack_t<sizeof...(Ts)>, util::decay_unwrap_t<Ts>...> bind_front(
```

Function template *bind_front* generates a forwarding call wrapper for *f*. Calling this wrapper is equivalent to invoking *f* with its (1) first or (2) last *sizeof...(Ts)* parameters bound to *vs*.

Parameters

- **f** – Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments
- **vs** – list of the arguments to bind to the (1) first or (2) last *sizeof...(Ts)* parameters of *f*

Returns A function object of type *T* that is unspecified, except that the types of objects returned by two calls to [hpx::bind_front](#) with the same arguments are the same.

```
template<typename F>  
constexpr std::decay_t<F> bind_front(F &&f)
```

namespace **serialization**

Functions

```
template<typename Archive, typename F, typename ...Ts>  
void serialize(Archive &ar, ::hpx::detail::bound_front<F, Ts...> &bound, unsigned int const version =  
0)
```

namespace **traits**

namespace **util**

Functions

```
template<typename F, typename... Ts> HPX_DEPRECATED_V (1, 8,
"hpx::util::bind_front is deprecated,
use hpx::bind_front instead") const expr decltype(auto) bind_front(F &&f
```

`hpx/functional/function.hpp`

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

Defines

`HPX_UTIL_REGISTER_FUNCTION_DECLARATION`(Sig, F, Name)

`HPX_UTIL_REGISTER_FUNCTION`(Sig, F, Name)

namespace `hpx`

Top level namespace.

`template<typename Sig, bool Serializable = false>`

class `function`

`#include <function.hpp>` Class template `hpx::function` is a general-purpose polymorphic function wrapper. Instances of `hpx::function` can store, copy, and invoke any CopyConstructible Callable target — functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members. The stored callable object is called the target of `hpx::function`. If an `hpx::function` contains no target, it is called empty. Invoking the target of an empty `hpx::function` results in `hpx::bad_function_call` exception being thrown. `hpx::function` satisfies the requirements of CopyConstructible and CopyAssignable.

`template<typename R, typename ...Ts, bool Serializable>`

class `function<R(Ts...), Serializable>` : public `util::detail::basic_function<R(Ts...), true, Serializable>`

Public Types

`using result_type = R`

Public Functions

`inline constexpr function(std::nullptr_t = nullptr) noexcept`

`function(function const&) = default`

`function(function&&) noexcept = default`

`function &operator=(function const&) = default`

```
function &operator=(function&&) noexcept = default

template<typename F, typename FD = std::decay_t<F>, typename Enable1 =
std::enable_if_t<!std::is_same_v<FD, function>>, typename Enable2 =
std::enable_if_t<is_invocable_r_v<R, FD&, Ts...>>>
inline function(F &&f)

template<typename F, typename FD = std::decay_t<F>, typename Enable1 =
std::enable_if_t<!std::is_same_v<FD, function>>, typename Enable2 =
std::enable_if_t<is_invocable_r_v<R, FD&, Ts...>>>
inline function &operator=(F &&f)
```

Private Types

```
using base_type = util::detail::basic_function<R(Ts...), true, Serializable>
```

```
namespace distributed
```

TypeDefs

```
template<typename Sig>
using function = hpx::function<Sig, true>
```

```
namespace util
```

TypeDefs

```
typedef hpx::function<Sig, Serializable> instead
```

hpx/functional/function_ref.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

Top level namespace.

```
template<typename Sig>
```

```
class function_ref
```

```
#include <function_ref.hpp> function_ref class is a vocabulary type with reference semantics for passing entities to call.
```

An example use case that benefits from higher-order functions is `retry(n, f)` which attempts to call `f` up to `n` times synchronously until success. This example might model the real-world scenario of repeatedly querying a flaky web service.

```
using payload = std::optional< /* ... */ >;
// Repeatedly invokes `action` up to `times` repetitions.
// Immediately returns if `action` returns a valid `payload`.
// Returns `std::nullopt` otherwise.
payload retry(size_t times, /* ????? */ action);
```

The passed-in action should be a callable entity that takes no arguments and returns a payload. This can be done with function pointers, [hpx::function](#) or a template but it is much simpler with [function_ref](#) as seen below:

```
payload retry(size_t times, function\_ref<payload\(\)> action);
```

```
template<typename R, typename ...Ts>
class function\_ref<R(Ts...)>
```

Public Functions

```
template<typename F, typename FD = std::decay_t<F>, typename Enable =
std::enable_if_t<!std::is_same_v<FD, function\_ref> && is\_invocable\_r\_v<R, F&, Ts...>>>
inline function\_ref(F &&f)

inline function\_ref(function\_ref const &other) noexcept

template<typename F, typename FD = std::decay_t<F>, typename Enable =
std::enable_if_t<!std::is_same_v<FD, function\_ref> && is\_invocable\_r\_v<R, F&, Ts...>>>
inline function\_ref &operator=(F &&f)

inline function\_ref &operator=(function\_ref const &other) noexcept

template<typename F, typename T = std::remove_reference_t<F>, typename Enable =
std::enable_if_t<!std::is_pointer_v<T>>>
inline void assign(F &&f)

template<typename T>
inline void assign(std::reference_wrapper<T> f_ref) noexcept

template<typename T>
inline void assign(T *f_ptr) noexcept

inline void swap(function\_ref &f) noexcept

inline R operator()(Ts... vs) const

inline std::size_t get\_function\_address() const

inline char const *get\_function\_annotation() const

inline util::itt::string_handle get\_function\_annotation\_itt() const
```

Protected Attributes

R (***vptr**)(void*, *Ts*&&...)

void ***object**

Private Types

```
using VTable = util::detail::function_ref_vtable<R(Ts...)>
```

Private Static Functions

```
template<typename T>
static inline VTable const *get_vtable() noexcept
```

namespace **util**

hpx/functional/invoke.hpp

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_INVOKE_R(*R*, *F*, ...)

namespace **hpx**

Top level namespace.

Functions

```
template<typename F, typename ...Tsutil::invoke_result<F, Ts...>::type invoke(F &&f, Ts&&... vs)
    noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(F,
        f), HPX_FORWARD(Ts, vs...))))
```

Invokes the given callable object *f* with the content of the argument pack *vs*

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

Throws `std::exception` – like objects thrown by call to object *f* with the argument types *vs*.

Returns The result of the callable object when it's called with the given argument types.

```
template<typename R, typename F, typename ...Ts>
constexpr R invoke_r(F &&f, Ts&&... vs) noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(F,f),
    HPX_FORWARD(Ts, vs)...)))
```

Invokes the given callable object f with the content of the argument pack vs

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

Throws `std::exception` – like objects thrown by call to object f with the argument types vs.

Template Parameters **R** – The result type of the function when it's called with the content of the given argument types vs.

Returns The result of the callable object when it's called with the given argument types.

namespace **functional**

struct **invoke**

Public Functions

```
template<typename F, typename ...Ts>
inline constexpr util::invoke_result<F, Ts...>::type operator()(F &&f, Ts&&... vs) const noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(F,f), HPX_FORWARD(Ts, vs)...)))
```

template<typename R>

struct **invoke_r**

Public Functions

```
template<typename F, typename ...Ts>
inline constexpr R operator()(F &&f, Ts&&... vs) const noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(F,f),
    HPX_FORWARD(Ts, vs)...)))
```

hpx/functional/invoke_fused.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace hpx

Top level namespace.

Functions

```
template<typename F, typename Tuple>
constexpr detail::invoke_result<F, Tuple>::type invoke_fused(F &&f, Tuple &&t) noexcept
    noexcept(detail::invoke_fused_impl(detail::fused_index_pack_t<Tuple>{}, HPX_FORWARD(F, f),
                                         HPX_FORWARD(Tuple, t))))
```

Invokes the given callable object f with the content of the sequenced type t (tuples, pairs).

Note: This function is similar to `std::apply` (C++17). The difference between `hpx::invoke` and `hpx::invoke_fused` is that the later unpacks the tuples while the former cannot. Turning a tuple into a parameter pack is not a trivial operation which makes `hpx::invoke_fused` rather useful.

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`.

Throws `std::exception` – like objects thrown by call to object f with the arguments contained in the sequenceable type t.

Returns The result of the callable object when it's called with the content of the given sequenced type.

```
template<typename R, typename F, typename Tuple>
constexpr R invoke_fused_r(F &&f, Tuple &&t) noexcept
    noexcept(detail::invoke_fused_impl(detail::fused_index_pack_t<Tuple>{}, HPX_FORWARD(F, f),
                                         HPX_FORWARD(Tuple, t))))
```

Invokes the given callable object f with the content of the sequenced type t (tuples, pairs).

Note: This function is similar to `std::apply` (C++17). The difference between `hpx::invoke` and `hpx::invoke_fused` is that the later unpacks the tuples while the former cannot. Turning a tuple into a parameter pack is not a trivial operation which makes `hpx::invoke_fused` rather useful.

Note: The difference between `hpx::invoke_fused` and `hpx::invoke_fused_r` is that the later allows to specify the return type as well.

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`.

Throws `std::exception` – like objects thrown by call to object `f` with the arguments contained in the sequenceable type `t`.

Template Parameters R – The result type of the function when it's called with the content of the given sequenced type.

Returns The result of the callable object when it's called with the content of the given sequenced type.

hpx/functional/mem_fn.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace hpx

Top level namespace.

Functions

```
template<typename M, typename C>
constexpr detail::mem_fn<M C::*> mem_fn(M C::* pm)
```

Function template `hpx::mem_fn` generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a `hpx::mem_fn`.

Parameters `pm` – pointer to member that will be wrapped

Returns a call wrapper of unspecified type with the following member:

```
template <typename... Ts>
constexpr typename util::invoke_result<MemberPointer, Ts...>::type
operator()(Ts&&... vs) noexcept;
```

Let `fn` be the call wrapper returned by a call to `hpx::mem_fn` with a pointer to member `pm`. Then the expression `fn(t, a2, ..., aN)` is equivalent to `HPX_INVOKE(pm, t, a2, ..., aN)`. Thus, the return type of `operator()` is `std::result_of<decltype(pm)(Ts&&...)>::type` or equivalently `std::invoke_result_t<decltype(pm), Ts&&...>`, and the value in `noexcept` specifier is equal to `std::is_nothrow_invocable_v<decltype(pm), Ts&&...>`. Each argument in `vs` is perfectly forwarded, as if by `std::forward<Ts>(vs)...`.

```
template<typename R, typename C, typename ...Ps>
constexpr detail::mem_fn<R (C::*)(Ps...)> mem_fn(R (C::* pm)(Ps...))
```

Function template `hpx::mem_fn` generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a `hpx::mem_fn`.

Parameters `pm` – pointer to member that will be wrapped

Returns a call wrapper of unspecified type with the following member:

```
template <typename... Ts>
constexpr typename util::invoke_result<MemberPointer, Ts...>::type
operator()(Ts&&... vs) noexcept;
```

Let `fn` be the call wrapper returned by a call to `hpx::mem_fn` with a pointer to member `pm`. Then the expression `fn(t,a2,...,aN)` is equivalent to `HPX_INVOKE(pm,t,a2,...,aN)`. Thus, the return type of operator() is `std::result_of<decltype(pm)(Ts&&...)>::type` or equivalently `std::invoke_result_t<decltype(pm), Ts&&...>`, and the value in `noexcept` specifier is equal to `std::is_nothrow_invocable_v<decltype(pm), Ts&&...>`. Each argument in `vs` is perfectly forwarded, as if by `std::forward<Ts>(vs)...`.

```
template<typename R, typename C, typename ...Ps>
constexpr detail::mem_fn<R(C::*)(Ps...) const> mem_fn(R(C::* pm)(Ps...) const)
```

Function template `hpx::mem_fn` generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a `hpx::mem_fn`.

Parameters `pm` – pointer to member that will be wrapped

Returns a call wrapper of unspecified type with the following member:

```
template <typename... Ts>
constexpr typename util::invoke_result<MemberPointer, Ts...>::type
operator()(Ts&&... vs) noexcept;
```

Let `fn` be the call wrapper returned by a call to `hpx::mem_fn` with a pointer to member `pm`. Then the expression `fn(t,a2,...,aN)` is equivalent to `HPX_INVOKE(pm,t,a2,...,aN)`. Thus, the return type of operator() is `std::result_of<decltype(pm)(Ts&&...)>::type` or equivalently `std::invoke_result_t<decltype(pm), Ts&&...>`, and the value in `noexcept` specifier is equal to `std::is_nothrow_invocable_v<decltype(pm), Ts&&...>`. Each argument in `vs` is perfectly forwarded, as if by `std::forward<Ts>(vs)...`.

hpx/functional/move_only_function.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

Defines

```
HPX_UTIL_REGISTER_UNIQUE_FUNCTION_DECLARATION(Sig, F, Name)
```

```
HPX_UTIL_REGISTER_UNIQUE_FUNCTION(Sig, F, Name)
```

namespace hpx

Top level namespace.

```
template<typename Sig, bool Serializable = false>
```

```
class move_only_function
```

```
#include <move_only_function.hpp> Class template hpx::move_only_function is a general-purpose polymorphic function wrapper. hpx::move_only_function objects can store and invoke any constructible (not required to be move constructible) Callable target &#8212; functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to member objects. The stored callable object is called the target of hpx::move_only_function. If an hpx::move_only_function contains no target, it is called empty. Unlike hpx::function, invoking an empty hpx::move_only_function results
```

in undefined behavior. `hpx::move_only_functions` supports every possible combination of cv-qualifiers, ref-qualifiers, and noexcept-specifiers not including volatile provided in its template parameter. These qualifiers and specifier (if any) are added to its operator(). `hpx::move_only_function` satisfies the requirements of MoveConstructible and MoveAssignable, but does not satisfy CopyConstructible or CopyAssignable.

```
template<typename R, typename ...Ts, bool Serializable>
class move_only_function<R(Ts...), Serializable> : public util::detail::basic_function<R(Ts...), false, Serializable>
```

Public Types

```
using result_type = R
```

Public Functions

```
inline constexpr move_only_function(std::nullptr_t = nullptr) noexcept
move_only_function(move_only_function const&) = delete
move_only_function(move_only_function&&) noexcept = default
move_only_function &operator=(move_only_function const&) = delete
move_only_function &operator=(move_only_function&&) noexcept = default

template<typename F, typename FD = std::decay_t<F>, typename Enable1 =
std::enable_if_t<!std::is_same_v<FD, move_only_function>>, typename Enable2 =
std::enable_if_t<is_invocable_r_v<R, FD&, Ts...>>>
inline move_only_function(F &&f)

template<typename F, typename FD = std::decay_t<F>, typename Enable1 =
std::enable_if_t<!std::is_same_v<FD, move_only_function>>, typename Enable2 =
std::enable_if_t<is_invocable_r_v<R, FD&, Ts...>>>
inline move_only_function &operator=(F &&f)
```

Private Types

```
using base_type = util::detail::basic_function<R(Ts...), false, Serializable>
```

```
namespace distributed
```

Typedefs

```
template<typename Sig>
using move_only_function = hpx::move_only_function<Sig, true>

namespace util
```

hpx/functional/traits/is_bind_expression.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace hpx

Top level namespace.

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>
```

```
#include <is_bind_expression.hpp> If T is the type produced by a call to std::bind, this template is derived from std::true_type. For any other type, this template is derived from std::false_type. This template may be specialized for a user-defined type T to implement UnaryTypeTrait with base characteristic of std::true_type to indicate that T should be treated by std::bind as if it were the type of a bind subexpression: when a bind-generated function object is invoked, a bound argument of this type will be invoked as a function object and will be given all the unbound arguments passed to the bind-generated object.
```

Subclassed by `hpx::is_bind_expression< T const >`

template<typename T>

```
struct is_bind_expression<T const> : public hpx::is_bind_expression<T>
```

namespace traits

Typedefs

```
typedef hpx::is_bind_expression<T> instead
```

Functions

```
template<typename T> HPX_DEPRECATED_V (1, 8,
"hpx::traits::is_bind_expression_v is deprecated,
use \"hpx::is_bind_expression_v instead\") inline const expr bool is_bind_expression_v
```

hpx/functional/traits/is_placeholder.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

Top level namespace.

```
template<typename T>
```

```
struct is_placeholder
```

```
#include <is_placeholder.hpp> If T is a standard, Boost, or HPX placeholder (_1, _2, _3, ...) then this
template is derived from std::integral_constant<int, 1>, std::integral_constant<int,
2>, std::integral_constant<int, 3>, respectively. Otherwise it is derived from ,
std::integral_constant<int, 0>.
```

```
namespace traits
```

Functions

```
template<typename T> HPX_DEPRECATED_V (1, 8,
"hpx::traits::is_placeholder_v is deprecated,
use \"hpx::is_placeholder_v instead\") inline const expr bool is_placeholder_v
```

futures

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

hpx/futures/future.hpp

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**

Top level HPX namespace.

Functions

```
template<typename R, typename U>
hpx::future<R> make_future(hpx::future<U> &&f)
```

Converts any future of type U to any other future of type R based on an existing conversion path from U to R.

```
template<typename R, typename U, typename Conv>
hpx::future<R> make_future(hpx::future<U> &&f, Conv &&conv)
```

Converts any future of type U to any other future of type R based on a given conversion function: R conv(U).

```
template<typename R, typename U>
hpx::future<R> make_future(hpx::shared_future<U> f)
```

Converts any *shared_future* of type U to any other future of type R based on an existing conversion path from U to R.

```
template<typename R, typename U, typename Conv>
hpx::future<R> make_future(hpx::shared_future<U> f, Conv &&conv)
```

Converts any future of type U to any other future of type R based on an existing conversion path from U to R.

```
template<typename R>
hpx::shared_future<R> make_shared_future(hpx::future<R> &&f) noexcept
```

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

```
template<typename R>
hpx::shared_future<R> &make_shared_future(hpx::shared_future<R> &f) noexcept
```

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

```
template<typename R>
hpx::shared_future<R> &&make_shared_future(hpx::shared_future<R> &&f) noexcept
```

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

```
template<typename R>
hpx::shared_future<R> const &make_shared_future(hpx::shared_future<R> const &f) noexcept
```

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

```
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,
Ts&&...
ts)

Creates a pre-initialized future object with allocator (extension)

```
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)
```

The function creates a shared state that is immediately ready and returns a future associated with that shared state. For the returned future, valid() == true and is_ready() == true.

```
template<int DeductionGuard = 0, typename Allocator, typename T>
future<hpx::util::decay_unwrap_t<T>> make_ready_future_alloc(Allocator const &a, T &&init)
```

```
template<int DeductionGuard = 0, typename T>
```

`future<hpx::util::decay_unwrap_t<T>> make_ready_future(T &&init)`

The function creates a shared state that is immediately ready and returns a future associated with that shared state. For the returned future, `valid() == true` and `is_ready() == true`.

template<typename T>

`future<T> make_exceptional_future(std::exception_ptr const &e)`

Creates a pre-initialized future object which holds the given error (extension)

template<typename T, typename E>

`future<T> make_exceptional_future(E e)`

Creates a pre-initialized future object which holds the given error (extension)

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)`

Creates a pre-initialized future object which gets ready at a given point in time (extension)

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 &&init)`

Creates a pre-initialized future object which gets ready after a given point in time (extension)

template<typename Allocator>

inline `future<void> make_ready_future_alloc(Allocator const &a)`

`future<void> make_ready_future()`

The function creates a shared state that is immediately ready and returns a future associated with that shared state. For the returned future, `valid() == true` and `is_ready() == true`.

inline `future<void> make_ready_future_at(hpx::chrono::steady_time_point const &abs_time)`

Creates a pre-initialized future object which gets ready at a given point in time (extension)

inline `future<void> make_ready_future_after(hpx::chrono::steady_duration const &rel_time)`

Creates a pre-initialized future object which gets ready after a given point in time (extension)

template<typename R>

class `future` : public `hpx::lcos::detail::future_base<future<R>, R>`

#include <`future_fwd.hpp`> The class template `hpx::future` provides a mechanism to access the result of asynchronous operations:

- An asynchronous operation (created via `hpx::async`, `hpx::packaged_task`, or `hpx::promise`) can provide a `hpx::future` object to the creator of that asynchronous operation.
- The creator of the asynchronous operation can then use a variety of methods to query, wait for, or extract a value from the `hpx::future`. These methods may block if the asynchronous operation has not yet provided a value.
- When the asynchronous operation is ready to send a result to the creator, it can do so by modifying shared state (e.g. `hpx::promise::set_value`) that is linked to the creator's `hpx::future`. Note that `hpx::future` references shared state that is not shared with any other asynchronous return objects (as opposed to `hpx::shared_future`).

Public Types

```
using result_type = R
```

```
using shared_state_type = typename base_type::shared_state_type
```

Public Functions

```
constexpr future() noexcept = default
```

```
future(future &&other) noexcept = default
```

```
inline future(future<future> &&other) noexcept
```

```
inline future(future<shared_future<R> &&other) noexcept
```

```
template<typename T>
```

```
inline future(future<T> &&other, std::enable_if_t<std::is_void_v<R> && !traits::is_future_v<T>,  
T* = nullptr) noexcept
```

```
~future() = default
```

```
future &operator=(future &&other) noexcept = default
```

```
inline shared_future<R> share() noexcept
```

```
inline hpx::traits::future_traits<future>::result_type get()
```

```
inline hpx::traits::future_traits<future>::result_type get(error_code &ec)
```

```
template<typename F>
```

```
inline decltype(auto) then(F &&f, error_code &ec = throws)
```

```
template<typename T0, typename F>
```

```
inline decltype(auto) then(T0 &&t0, F &&f, error_code &ec = throws)
```

```
template<typename Allocator, typename F>
```

```
inline auto then_alloc(Allocator const &alloc, F &&f, error_code &ec = throws) ->  
    decltype(base_type::then_alloc(alloc, HPX_MOVE(*this),  
        HPX_FORWARD(F, f), ec))
```

Private Types

```
using base_type = lcos::detail::future_base<future<R>, R>
```

Private Functions

```
inline explicit future(hpx::intrusive_ptr<shared_state_type> const &state)
inline explicit future(hpx::intrusive_ptr<shared_state_type> &&state)
template<typename SharedState>
inline explicit future(hpx::intrusive_ptr<SharedState> const &state)
```

Friends

```
friend struct hpx::traits::future_access
struct invalidate
```

Public Functions

```
inline explicit constexpr invalidate(future &f) noexcept
inline ~invalidate()
```

Public Members

future &*f_*

```
template<typename R>
class shared_future : public hpx::lcos::detail::future_base<shared_future<R>, R>
```

#include <future_fwd.hpp> The class template *hpx*::*shared_future* provides a mechanism to access the result of asynchronous operations, similar to *hpx*::*future*, except that multiple threads are allowed to wait for the same shared state. Unlike *hpx*::*future*, which is only moveable (so only one instance can refer to any particular asynchronous result), *hpx*::*shared_future* is copyable and multiple shared future objects may refer to the same shared state. Access to the same shared state from multiple threads is safe if each thread does it through its own copy of a *shared_future* object.

Public Types

using **result_type** = *R*

using **shared_state_type** = typename *base_type*::*shared_state_type*

Public Functions

```
constexpr shared_future() noexcept = default  
  
shared_future(shared_future const &other) = default  
  
shared_future(shared_future &&other) noexcept = default  
  
inline shared_future(future<R> &&other) noexcept  
  
inline shared_future(future<shared_future> &&other) noexcept  
  
template<typename T>  
inline 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() = default  
  
shared_future &operator=(shared_future const &other) = default  
  
shared_future &operator=(shared_future &&other) noexcept = default  
  
inline hpx::traits::future_traits<shared_future>::result_type get() const  
  
inline hpx::traits::future_traits<shared_future>::result_type get(error_code &ec) const  
  
template<typename F>  
inline decltype(auto) then(F &&f, error_code &ec = throws) const  
  
template<typename T0, typename F>  
inline decltype(auto) then(T0 &&t0, F &&f, error_code &ec = throws) const  
  
template<typename Allocator, typename F>  
inline auto then_alloc(Allocator const &alloc, F &&f, error_code &ec = throws) ->  
    decltype(base_type::then_alloc(alloc, HPX_MOVE(*this),  
        HPX_FORWARD(F, f), ec))
```

Private Types

```
using base_type = lcos::detail::future_base<shared_future<R>, R>
```

Private Functions

```
inline explicit shared_future(hpx::intrusive_ptr<shared_state_type> const &state)  
  
inline explicit shared_future(hpx::intrusive_ptr<shared_state_type> &&state)  
  
template<typename SharedState>  
inline explicit shared_future(hpx::intrusive_ptr<SharedState> const &state)
```

Friends

```
friend struct hpx::traits::future_access
```

```
namespace lcos
```

Functions

```
template<typename R, typename U> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_future is deprecated. Use hpx::make_future instead.") hpx
```

```
template<typename R, typename U, typename Conv> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_future is deprecated. Use hpx::make_future instead.") hpx
```

```
template<typename T, typename Allocator, typename... Ts> HPX_DEPRECATED_V (1,  
8, "hpx::lcos::make_ready_future_alloc is deprecated.  
Use \"hpx::make_ready_future_alloc instead.") std
```

```
template<typename T, typename... Ts> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_ready_future is deprecated.  
Use \"hpx::make_ready_future instead.") std
```

```
template<typename T> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_exceptional_future is deprecated.  
Use \"hpx::make_exceptional_future instead.") hpx
```

```
template<typename T, typename E> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_exceptional_future is deprecated.  
Use \"hpx::make_exceptional_future instead.") hpx
```

```
template<int DeductionGuard = 0, typename T> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_ready_future_at is deprecated.  
Use \"hpx::make_ready_future_at instead.") hpx
```

```
template<int DeductionGuard = 0, typename T> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_ready_future_after is deprecated.  
Use \"hpx::make_ready_future_after instead.") hpx
```

```
template<typename Allocator> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_ready_future_alloc is deprecated.  
Use \"hpx::make_ready_future_alloc instead.") hpx
```

```
template<typename T> HPX_DEPRECATED_V (1, 8,  
"hpx::lcos::make_ready_future is deprecated.  
Use \"hpx::make_ready_future instead.") std
```

```
template<typename T> HPX_DEPRECATED_V (1, 8,
"hpx::lcos::make_ready_future_at is deprecated.
Use \"hpx::make_ready_future_at instead.") std

template<typename T> HPX_DEPRECATED_V (1, 8,
"hpx::lcos::make_ready_future_after is deprecated.
Use \"hpx::make_ready_future_after instead.") std
```

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)
```

hpx/futures/future_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Top level HPX namespace.

```
template<typename R>
class future : public hpx::lcos::detail::future_base<future<R>, R>
{
    #include <future_fwd.hpp>
};

template<typename R>
class shared_future : public hpx::lcos::detail::future_base<shared_future<R>, R>
{
    #include <future_fwd.hpp>
};
```

namespace **lcos**

Typedefs

```
typedef hpx::future<R> instead
```

namespace **lcos**

hpx/futures/packaged_task.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
template<typename Sig, typename Allocator>
struct uses_allocator<hpx::packaged_task<Sig>, Allocator> : public true_type
```

namespace **hpx**

Top level HPX namespace.

```
template<typename Sig>
```

```
class packaged_task
```

```
#include <packaged_task.hpp> The class template hpx::packaged_task wraps any Callable` target (function, lambda expression, bind expression, or another function object) so that it can be invoked asynchronously. Its return value or exception thrown is stored in a shared state which can be accessed through hpx::future objects. Just like hpx::function, hpx::packaged_task is a polymorphic, allocator-aware container: the stored callable target may be allocated on heap or with a provided allocator.
```

```
template<typename R, typename ...Ts>
```

```
class packaged_task<R(Ts...)>
```

Public Functions

```
packaged_task() = default
```

```
template<typename F, typename FD = std::decay_t<F>, typename Enable =
std::enable_if_t<!std::is_same_v<FD, packaged_task> && is_invocable_r_v<R, FD&, Ts...>>>
inline explicit packaged_task(F &&f)
```

```
template<typename Allocator, typename F, typename FD = std::decay_t<F>, typename Enable =
std::enable_if_t<!std::is_same_v<FD, packaged_task> && is_invocable_r_v<R, FD&, Ts...>>>
inline explicit packaged_task(std::allocator_arg_t, Allocator const &a, F &&f)
```

```
packaged_task(packaged_task const &rhs) noexcept = delete
```

```
packaged_task(packaged_task &&rhs) noexcept = default
```

```
packaged_task &operator=(packaged_task const &rhs) noexcept = delete
```

```
packaged_task &operator=(packaged_task &&rhs) noexcept = default
```

```
inline void swap(packaged_task &rhs) noexcept
```

```
inline void operator()(Ts... ts)
```

```
inline hpx::future<R> get_future(error_code &ec = throws)
```

```
inline bool valid() const noexcept
```

```
inline void reset(error_code &ec = throws)
```

```
inline void set_exception(std::exception_ptr const &e)
```

Private Types

```
using function_type = hpx::move_only_function<R(Ts...)>
```

Private Members

```
function_type function_
```

```
hpx::promise<R> promise_
```

```
namespace lcos
```

```
namespace local
```

Typedefs

```
typedef hpx::packaged_task<Sig> instead
```

```
namespace std
```

Functions

```
template<typename Sig>
void swap(hpx::packaged_task<Sig> &lhs, hpx::packaged_task<Sig> &rhs)
```

```
template<typename Sig, typename Allocator> packaged_task< Sig >,
Allocator > : public true_type
```

hpx/futures/promise.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
template<typename R, typename Allocator>
```

```
struct uses_allocator<hpx::promise<R>, Allocator> : public true_type
```

```
namespace hpx
```

Top level HPX namespace.

Functions

```
template<typename R>
void swap(promise<R> &x, promise<R> &y) noexcept
```

```
template<typename R>
```

```
class promise : public hpx::detail::promise_base<R>
```

#include <promise.hpp> The class template *hpx::promise* provides a facility to store a value or an exception that is later acquired asynchronously via a *hpx::future* object created by the *hpx::promise* object. Note that the *hpx::promise* object is meant to be used only once. Each promise is associated with a shared state, which contains some state information and a result which may be not yet evaluated, evaluated to a value (possibly void) or evaluated to an exception. A promise may do three things with the shared state:

- make ready: the promise stores the result or the exception in the shared state. Marks the state ready and unblocks any thread waiting on a future associated with the shared state.
- release: the promise gives up its reference to the shared state. If this was the last such reference, the shared state is destroyed. Unless this was a shared state created by *hpx::async* which is not yet ready, this operation does not block.
- abandon: the promise stores the exception of type *hpx::future_error* with error code *hpx::future_errc::broken_promise*, makes the shared state ready, and then releases it. The promise is the “push” end of the promise-future communication channel: the operation that stores a value in the shared state synchronizes-with (as defined in *hpx::memory_order*) the successful return from any function that is waiting on the shared state (such as *hpx::future::get*). Concurrent access to the same shared state may conflict otherwise: for example multiple callers of *hpx::shared_future::get* must either all be read-only or provide external synchronization.

Public Functions

```
promise() = default
```

```
template<typename Allocator>
inline promise(std::allocator_arg_t, Allocator const &a)
```

```
promise(promise &&other) noexcept = default
```

```
~promise() = default
```

```
promise &operator=(promise &&other) noexcept = default
```

```
inline void swap(promise &other) noexcept
```

```
inline void set_value(R const &r)
```

```
inline void set_value(R &&r)
```

```
template<typename ...Ts>
```

```
inline void set_value(Ts&&... ts)
```

Private Types

```
using base_type = detail::promise_base<R>

template<typename R>
class promise<R&> : public hpx::detail::promise_base<R&>
```

Public Functions

```
promise() = default

template<typename Allocatorpromise(std::allocator_arg_t, Allocator const &a)
```

```
promise(promise &&other) noexcept = default
```

```
~promise() = default
```

```
promise &operator=(promise &&other) noexcept = default
```

```
inline void swap(promise &other) noexcept
```

```
inline void set_value(R &r)
```

Private Types

```
using base_type = detail::promise_base<R&>

template<>
class promise<void> : public hpx::detail::promise_base<void>
```

Public Functions

```
promise() = default

template<typename Allocatorpromise(std::allocator_arg_t, Allocator const &a)
```

```
promise(promise &&other) noexcept = default
```

```
~promise() = default
```

```
promise &operator=(promise &&other) noexcept = default
```

```
inline void swap(promise &other) noexcept
```

```
inline void set_value()
```

Private Types

```
using base_type = detail::promise_base<void>

namespace lcos

namespace local

namespace std

template<typename R, typename Allocator> promise< R >,
Allocator > : public true_type
```

io_service

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/io_service/io_service_pool.hpp

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*)

```
explicit 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]

```
explicit 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

```
inline 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.

```
inline 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

```
inline 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 &**notifier_**

call this for each thread start/stop

char const ***pool_name_**

char const ***pool_name_postfix_**

bool **waiting_**

Set to true if waiting for work to finish.

std::unique_ptr<*barrier*> **wait_barrier_**

std::unique_ptr<*barrier*> **continue_barrier_**

Icos_local

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/lcos_local/and_gate.hpp

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<hpx::no_mutex>
```

Public Functions

```
inline and_gate(std::size_t count = 0)
```

```
inline and_gate(and_gate &&rhs) noexcept
```

inline *and_gate* &**operator=**(*and_gate* &&rhs) noexcept

```
template<typename Lock>
```

```
inline 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>
```

```
inline 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>
```

inline bool **set**(*std::size_t which, Lock l, error_code &ec = throws*)

```
template<typename Lock>
```

```
inline 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<hpx::no_mutex> base_type

template<typename Mutex = hpx::spinlock>
struct base_and_gate
```

Public Functions

```
inline explicit 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.

inline base_and_gate(base_and_gate &&rhs) noexcept

inline base_and_gate &operator=(base_and_gate &&rhs) noexcept

inline hpx::future<void> get_future(std::size_t count = std::size_t(-1), std::size_t
    *generation_value = nullptr, error_code &ec =
    hpx::throws)
```

inline *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*)

inline bool **set**(*std::size_t* which, *error_code* &ec = *throws*)

inline 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.

```
template<typename Lock>
inline std::size_t next_generation(Lock &l, std::size_t new_generation)

inline std::size_t next_generation(std::size_t new_generation = std::size_t(-1))

template<typename Lock>
inline std::size_t generation(Lock &l) const

inline std::size_t generation() const
```

Protected Types

```
typedef Mutex mutex_type
```

Protected Functions

```
inline bool trigger_conditions(error_code &ec = throws)

template<typename OuterLock>
inline 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>
inline 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>
inline bool set(std::size_t which, OuterLock outer_lock, error_code &ec = throws)

    Set the data which has to go into the segment which.

inline bool test_condition(std::size_t generation_value)

template<typename Lock>
inline 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>
inline 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

```
mutable mutex_type mtx_
```

```
hpx::detail::dynamic_bitset received_segments_
```

```
hpx::promise<void> promise_
```

```
std::size_t generation_
```

```
condition_list_type conditions_
```

```
struct manage_condition
```

Public Functions

```
inline manage_condition(base_and_gate &gate, conditional_trigger &cond)
inline ~manage_condition()

template<typename Condition>
inline hpx::future<void> get_future(Condition &&func, error_code &ec = hpx::throws)
```

Public Members

base_and_gate &**this_**

condition_list_type::iterator **it_**

hpx/lcos_local/conditional_trigger.hpp

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 **conditional_trigger**

Public Functions

```
conditional_trigger() = default
conditional_trigger(conditional_trigger &&rhs) noexcept = default
conditional_trigger &operator=(conditional_trigger &&rhs) noexcept = default

template<typename Condition>
inline hpx::future<void> get_future(Condition &&func, error_code &ec = hpx::throws)
    get a future allowing to wait for the trigger to fire
inline void reset()
inline bool set(error_code &ec = throws)
    Trigger this object.
```

Private Members

`hpx::promise<void> promise_`

`hpx::function<bool()> cond_`

`hpx/lcos_local/trigger.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **lcos**

 namespace **local**

 template<typename **Mutex** = `hpx::spinlock`>

 struct **base_trigger**

Public Functions

 inline **base_trigger**()

 inline **base_trigger**(`base_trigger` &&rhs) noexcept

 inline `base_trigger` &**operator=**(`base_trigger` &&rhs) noexcept

 inline `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

 inline bool **set**(`error_code` &ec = `throws`)

 Trigger this object.

 inline 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.

 inline `std::size_t` **next_generation**()

 inline `std::size_t` **generation**() const

Protected Types

```
typedef Mutex mutex_type
```

Protected Functions

```
inline bool trigger_conditions(error_code &ec = throws)
```

```
template<typename Lock>
inline 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

```
inline bool test_condition(std::size_t generation_value)
```

Private Members

```
mutable mutex_type mtx_
```

```
hpx::promise<void> promise_
```

```
std::size_t generation_
```

```
condition_list_type conditions_
```

```
struct manage_condition
```

Public Functions

```
inline manage_condition(base_trigger &gate, conditional_trigger &cond)
```

```
inline ~manage_condition()
```

```
template<typename Condition>
```

```
inline hpx::future<void> get_future(Condition &&func, error_code &ec = hpx::throws)
```

Public Members

base_trigger &**this**_

condition_list_type::iterator **it**_

```
struct trigger : public hpx::lcos::local::base_trigger<hpx::no_mutex>
```

Public Functions

inline **trigger**()

inline **trigger**(*trigger* &&rhs) noexcept

inline *trigger* &**operator=**(*trigger* &&rhs) noexcept

template<typename **Lock**>

```
inline void synchronize(std::size_t generation_value, Lock &l, char const *function_name =  
    "trigger::synchronize", error_code &ec = throws)
```

Private Types

```
typedef base_trigger<hpx::no_mutex> base_type
```

pack_traversal

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/pack_traversal/pack_traversal.hpp

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> map_pack (Mapper &&mapper, T &&... pack)
```

Maps the pack with the given mapper.

This function tries to visit all plain elements which may be wrapped in:

- homogeneous containers (*std::vector*, *std::list*)
- heterogeneous containers (*hpx::tuple*, *std::pair*, *std::array*) and re-assembles the pack with the result of the mapper. Mapping from one type to a different one is supported.

Elements that aren't accepted by the mapper are routed through and preserved through the hierarchy.

```
// Maps all integers to floats
map_pack([](int value) {
    return float(value);
},
1, hpx::make_tuple(2, std::vector<int>{3, 4}), 5);
```

Throws `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.

Returns The mapped element or in case the pack contains multiple elements, the pack is wrapped into a `hpx::tuple`.

hpx/pack_traversal/pack_traversal_async.hpp

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 Visitor, typename ...T>
auto traverse_pack_async(Visitor &&visitor, T&&... pack) ->
    decltype(detail::apply_pack_transform_async(HPX_FORWARD(Visitor,
    visitor), HPX_FORWARD(T, pack)...))
```

Traverses the pack with the given visitor in an asynchronous way.

This function works in the same way as `traverse_pack`, however, we are able to suspend and continue the traversal at later time. Thus we require a visitor callable object which provides three `operator()` overloads as depicted by the code sample below:

```
struct my_async_visitor
{
    template <typename T>
    bool operator()(async_traverse_visit_tag, T&& element)
    {
        return true;
    }

    template <typename T, typename N>
    void operator()(async_traverse_detach_tag, T&& element, N&& next)
    {
    }
```

(continues on next page)

(continued from previous page)

```
template <typename T>
void operator()(async_traverse_complete_tag, T&& pack)
{
}
};
```

See `traverse_pack` for a detailed description about the traversal behavior and capabilities.

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.

Returns A `hpx::intrusive_ptr` that references an instance of the given visitor object.

```
template<typename Allocator, typename Visitor, typename ...T>
auto traverse_pack_async_allocator(Allocator const &alloc, Visitor &&visitor, T &&... pack) ->
    de-
    cltype(detail::apply_pack_transform_async_allocator(alloc,
        HPX_FORWARD(Visitor, visitor), HPX_FORWARD(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.

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.

Returns A hpx::intrusive_ptr that references an instance of the given visitor object.

hpx/pack_traversal/unwrap.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename ...Args>
auto unwrap(Args&&... args) -> decltype(util::detail::unwrap_depth_impl<1U>(HPX_FORWARD(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.

Parameters `args` – the arguments that are unwrapped which may contain any arbitrary future or non future type.

Throws `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.

Returns 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.

```
template<std::size_t Depth, typename ...Args>
auto unwrap_n(Args&&... args) ->
    decltype(util::detail::unwrap_depth_impl<Depth>(HPX_FORWARD(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 maxmally.

```
template<typename ...Args>
auto unwrap_all(Args&&... args) ->
    decltype(util::detail::unwrap_depth_impl<0U>(HPX_FORWARD(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) ->
    decltype(util::detail::functional_unwrap_depth_impl<1U>(HPX_FORWARD(T, callable)))
```

Returns a callable object which unwraps its arguments upon invocation using the hpx::unwrap() function and then passes the result to the given callable object.

```
auto callable = hpx::unwrapping([](int left, int right) {
    return left + right;
});

int i1 = callable(hpx::make_ready_future(1),
                  hpx::make_ready_future(2));
```

See hpx::unwrap() for a detailed description.

Parameters callable – the callable object which which is called with the result of the corresponding unwrap function.

```
template<std::size_t Depth, typename T>
auto unwrapping_n(T &&callable) ->
    decltype(util::detail::functional_unwrap_depth_impl<Depth>(HPX_FORWARD(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) ->
    decltype(util::detail::functional_unwrap_depth_impl<0U>(HPX_FORWARD(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.

preprocessor

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/preprocessor/cat.hpp

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

hpx/preprocessor/expand.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

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.

Example:

```
#define MACRO(a, b, c) (a)(b)(c)
#define ARGS() (1, 2, 3)

HPX_PP_EXPAND(MACRO ARGS()) // expands to (1)(2)(3)
```

Parameters

- **X** – Token to be expanded twice

hpx/preprocessor/nargs.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

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)
```

Expands to:

```
3
2
1
```

Parameters

- **...** – The variadic number of arguments

hpx/preprocessor/stringize.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

hpx/preprocessor/strip_parens.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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:

This produces the following output

```
no_parens
with_parens
```

Parameters

- `X` – Symbol to strip parens from

resiliency

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/resiliency/replay_executor.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

```
template<typename BaseExecutor, typename Validator>
struct is_two_way_executor<hpx::resiliency::experimental::replay_executor<BaseExecutor, Validator>> : public
true_type

template<typename BaseExecutor, typename Validator>
struct is_bulk_two_way_executor<hpx::resiliency::experimental::replay_executor<BaseExecutor, Validator>> :
public true_type

namespace hpx

    namespace parallel

        namespace execution

            template<typename BaseExecutor,
            typename Validator> replay_executor< BaseExecutor,
            Validator > > : public true_type

            template<typename BaseExecutor,
            typename Validator> replay_executor< BaseExecutor,
            Validator > > : public true_type

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

```
using execution_category = hpx::traits::executor_execution_category_t<BaseExecutor>

using executor_parameters_type = hpx::traits::executor_parameters_type_t<BaseExecutor>

template<typename Result>

using future_type = hpx::traits::executor_future_t<BaseExecutor, Result>
```

Public Functions

```
template<typename F>
inline explicit replay_executor(BaseExecutor &exec, std::size_t n, F &&f)

inline bool operator==(replay_executor const &rhs) const noexcept

inline bool operator!=(replay_executor const &rhs) const noexcept

inline replay_executor const &context() const noexcept
```

Public Static Attributes

```
static constexpr int num_spread = 4
```

```
static constexpr int num_tasks = 128
```

Private Functions

```
template<typename F, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::async_execute_t,
replay_executor const &exec, F &&f, Ts&&... ts)

template<typename F, typename S, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::bulk_async_execute_t,
replay_executor const &exec, F &&f, S const
&shape, Ts&&... ts)
```

Private Members

BaseExecutor &exec_

std::size_t replay_count_

Validate validator_

hpx/resiliency/replicate_executor.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
template<typename BaseExecutor, typename Voter, typename Validator>
struct is_two_way_executor<hpx::resiliency::experimental::replicate_executor<BaseExecutor, Voter,
Validator>> : public true_type

template<typename BaseExecutor, typename Voter, typename Validator>
struct is_bulk_two_way_executor<hpx::resiliency::experimental::replicate_executor<BaseExecutor, Voter,
Validator>> : public true_type

namespace hpx

    namespace parallel

        namespace execution

            template<typename BaseExecutor, typename Voter,
            typename Validator> replicate_executor< BaseExecutor, Voter,
            Validator > > : public true_type

            template<typename BaseExecutor, typename Voter,
            typename Validator> replicate_executor< BaseExecutor, Voter,
            Validator > > : public true_type

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_replicate_executor(BaseExecutor, Voter, Validate)
```

```
template<typename BaseExecutor, typename Validate>
```

```
replicate_executor<BaseExecutor, detail::replicate_voter, typename std::decay<Validate>::type> make_replicate_executor(BaseExecutor, detail::replicate_voter, Validate)
```

```
template<typename BaseExecutor>
```

```
replicate_executor<BaseExecutor, detail::replicate_voter, detail::replicate_validator> make_replicate_executor(BaseExecutor)
```

```
&
```

```
std::shared_ptr<detail::replicate_executor<BaseExecutor, detail::replicate_voter, detail::replicate_validator>> make_replicate_executor(BaseExecutor)
```

```
template<typename BaseExecutor, typename Vote, typename Validate>
```

```
class replicate_executor
```

Public Types

```
using execution_category = hpx::traits::executor_execution_category_t<BaseExecutor>
```

```
using executor_parameters_type = hpx::traits::executor_parameters_type_t<BaseExecutor>
```

```
template<typename Result>
```

```
using future_type = hpx::traits::executor_future_t<BaseExecutor, Result>
```

Public Functions

```
template<typename V, typename F>
inline explicit replicate_executor(BaseExecutor &exec, std::size_t n, V &&v, F &&f)  
  
inline bool operator==(replicate_executor const &rhs) const noexcept  
  
inline bool operator!=(replicate_executor const &rhs) const noexcept  
  
inline replicate_executor const &context() const noexcept
```

Public Static Attributes

```
static constexpr int num_spread = 4
```

```
static constexpr int num_tasks = 128
```

Private Functions

```
template<typename F, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::async_execute_t,
                                         replicate_executor const &exec, F &&f, Ts&&... ts)  
  
template<typename F, typename S, typename ...Ts>
inline decltype(auto) friend tag_invoke(hpx::parallel::execution::bulk_async_execute_t,
                                         replicate_executor const &exec, F &&f, S const
                                         &shape, Ts&&... ts)
```

Private Members

BaseExecutor &**exec_**

std::size_t **replicate_count_**

Vote **voter_**

Validate **validator_**

runtime_configuration

See *Public API* for a list of names and headers that are part of the public HPX API.

[hpx/runtime_configuration/component_commandline_base.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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**() = default

virtual *hpx*::program_options::options_description **add_commandline_options**() = 0

Return any additional command line options valid for this component.

Note: This function will be executed by the runtime system during system startup.

Returns The module is expected to fill a `options_description` object with any additional command line options this component will handle.

[hpx/runtime_configuration/component_registry_base.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

inline 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.

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.

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

virtual void register_component_type() = 0

 Return the unique identifier of the component type this factory is responsible for.

Parameters

- **locality** – [in] The id of the locality this factory is responsible for.
- **agas_client** – [in] The AGAS client to use for component id registration (if needed).

Returns Returns the unique identifier of the component type this factory instance is responsible for. This function throws on any error.

hpx/runtime_configuration/plugin_registry_base.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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

inline 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

Parameters **fillini** – [in, out] The module is expected to fill this vector with the information (one line per vector element) for all plugins implemented in this module.

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

inline virtual void **init**(int*, char***, *util*::runtime_configuration&)

hpx/runtime_configuration/runtime_mode.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Enums

enum class **runtime_mode**

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

Values:

enumerator **invalid**

enumerator `console`

The runtime is the console locality.

enumerator `worker`

The runtime is a worker locality.

enumerator `connect`

The runtime is a worker locality connecting late

enumerator `local`

The runtime is fully local.

enumerator `default_`

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

enumerator `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

`runtime_local`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

`hpx/runtime_local/component_startup_shutdown_base.hpp`

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**() = default

virtual bool **get_startup_function**(*startup_function_type* &*startup*, bool &*pre_startup*) = 0

Return any startup function for this component.

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.

Returns Returns *true* if the parameter *startup* has been successfully initialized with the startup function.

virtual bool **get_shutdown_function**(*shutdown_function_type* &*shutdown*, bool &*pre_shutdown*) = 0

Return any shutdown function for this component.

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.

Returns Returns *true* if the parameter *shutdown* has been successfully initialized with the shutdown function.

hpx/runtime_local/custom_exception_info.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

See also:

`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`.

Throws `std::bad_alloc` – (if any of the required allocation operations fail)

Returns The formatted string holding all of the available diagnostic information stored in the given exception instance.

`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.

See also:

`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`.

Throws nothing –

Returns 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`.

`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.

See also:

<code>hpx::diagnostic_information()</code>	<code>hpx::get_error_process_id()</code>	<code>hpx::get_error_function_name()</code>
<code>hpx::get_error_file_name()</code>	<code>hpx::get_error_line_number()</code>	<code>hpx::get_error_os_thread()</code>
<code>hpx::get_error_thread_id()</code>	<code>hpx::get_error_thread_description()</code>	<code>hpx::get_error()</code>
<code>hpx::get_error_backtrace()</code>	<code>hpx::get_error_env()</code>	<code>hpx::get_error_what()</code>
<code>hpx::get_error_state()</code>		<code>hpx::get_error_config()</code>

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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns 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.

`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.

See also:

<code>hpx::diagnostic_information()</code>	<code>hpx::get_error_host_name()</code>	<code>hpx::get_error_function_name()</code>
<code>hpx::get_error_file_name()</code>	<code>hpx::get_error_line_number()</code>	<code>hpx::get_error_os_thread()</code>
<code>hpx::get_error_thread_id()</code>	<code>hpx::get_error_thread_description()</code>	<code>hpx::get_error()</code>
<code>hpx::get_error_backtrace()</code>	<code>hpx::get_error_env()</code>	<code>hpx::get_error_what()</code>
<code>hpx::get_error_state()</code>		<code>hpx::get_error_config()</code>

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`.

Throws nothing –

Returns 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.

`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.

See also:

`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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns 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.

`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.

See also:

`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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns 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.

`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.

See also:

`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`.

Throws nothing –

Returns The sequence number of the OS-thread used to execute the HPX-thread from which the exception was thrown. If the exception instance does not hold this information, the function will return `std::size(-1)`.

`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.

See also:

`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`.

Throws nothing –

Returns 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(0)`.

`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.

See also:

`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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns 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.

`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.

See also:

<code>hpx::diagnostic_information()</code> ,	<code>hpx::get_error_host_name()</code> ,	<code>hpx::get_error_process_id()</code> ,
<code>hpx::get_error_function_name()</code> ,	<code>hpx::get_error_file_name()</code> ,	<code>hpx::get_error_line_number()</code> ,
<code>hpx::get_error_os_thread()</code> ,	<code>hpx::get_error_thread_id()</code> ,	<code>hpx::get_error_backtrace()</code> ,
<code>hpx::get_error_env()</code> ,	<code>hpx::get_error()</code> ,	<code>hpx::get_error_state()</code>
<code>hpx::get_error_thread_description()</code>		<code>hpx::get_error_what()</code> ,

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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns 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.

`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.

See also:

<code>hpx::diagnostic_information()</code> ,	<code>hpx::get_error_host_name()</code> ,	<code>hpx::get_error_process_id()</code> ,
<code>hpx::get_error_function_name()</code> ,	<code>hpx::get_error_file_name()</code> ,	<code>hpx::get_error_line_number()</code> ,
<code>hpx::get_error_os_thread()</code> ,	<code>hpx::get_error_thread_id()</code> ,	<code>hpx::get_error_backtrace()</code> ,
<code>hpx::get_error_env()</code> ,	<code>hpx::get_error()</code> ,	<code>hpx::get_error_what()</code> ,
<code>hpx::get_error_thread_description()</code>		<code>hpx::get_error_state()</code>

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`.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns 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.

hpx/runtime_local/get_locality_id.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

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.

hpx/runtime_local/get_locality_name.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

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.

See also:

`future<std::string> get_locality_name(hpx::id_type const& id)`

Returns 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.

hpx/runtime_local/get_num_all_localities.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

See also:

`hpx::find_all_localities`, `hpx::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`.

`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.

See also:

`hpx::find_all_localities`, `hpx::get_num_localities`

Note: This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

`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.

See also:

`hpx::find_all_localities`, `hpx::get_num_localities`

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*.

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.

hpx/runtime_local/get_os_thread_count.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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 **threads**

hpx/runtime_local/get_thread_name.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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>”.

hpx/runtime_local/get_worker_thread_num.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/runtime_local/report_error.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

hpx/runtime_local/runtime_local.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
void set_error_handlers()
```

```
class runtime
```

Public Types

```
using notification_policy_type = threads::policies::callback_notifier
```

Generate a new notification policy instance for the given thread name prefix

```
using hpx_main_function_type = int()
```

The *hpx_main_function_type* is the default function type usable as the main HPX thread function.

```
using hpx_errorsink_function_type = void(std::uint32_t, std::string const&)
```

Public Functions

```

virtual notification_policy_type get_notification_policy(char const *prefix,
                                                    runtime_local::os_thread_type type)

state get_state() const
void set_state(state s)

explicit runtime(hpx::util::runtime_configuration &rtecfg, 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(hpx::function<void()> const &f)
    Manage list of functions to call on exit.

void starting()
    Manage runtime ‘stopped’ state.

void stopping()
    Call all registered on_exit functions.

bool stopped() const
    This accessor returns whether the runtime instance has been stopped.

hpx::util::runtime_configuration &get_config()
    access configuration information

hpx::util::runtime_configuration const &get_config() const

std::size_t get_instance_number() const

util::thread_mapper &get_thread_mapper()
    Return a reference to the internal PAPI thread manager.

threads::topology const &get_topology() const

virtual int run(hpx::function<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.

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.

Returns This function will return the value as returned as the result of the invocation of the function object given by the parameter *func*.

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.

Returns This function will always return 0 (zero).

virtual void **rethrow_exception()**

Rethrow any stored exception (to be called after [stop\(\)](#))

virtual int **start(*hpx::function<hpx_main_function_type>*** const &func, bool blocking = false)

Start the runtime system.

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.

Returns 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.

virtual int **start(bool blocking = false)**

Start the runtime system.

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 .

Returns 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.

virtual int **wait()**

Wait for the shutdown action to be executed.

Returns 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:

Note: This function should be called for each thread exactly once. It will fail if it is called more than once.

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.

Returns 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`).

Returns 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(hpx::function<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
inline virtual std::uint32_t assign_cores(std::string const&, std::uint32_t)
inline 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<hpx::function<void()>>
```

Protected Functions

```

explicit runtime(hpx::util::runtime_configuration &rtcfg)
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(hpx::function<runtime::hpx_main_function_type> const
                           &func, int &result, bool call_startup_functions)

void wait_helper(mutex &mtx, condition_variable &cond, bool &running)

```

Protected Attributes

```
on_exit_type on_exit_functions_

mutable std::mutex mtx_

hpx::util::runtime_configuration rtcfg_

long instance_number_

std::unique_ptr<util::thread_mapper> thread_support_

threads::topology &topology_

std::atomic<state> state_

notification_policy_type::on_startstop_type on_start_func_

notification_policy_type::on_startstop_type on_stop_func_

notification_policy_type::on_error_type on_error_func_

int result_

std::exception_ptr exception_

notification_policy_type main_pool_notifier_

util::io_service_pool main_pool_

notification_policy_type notifier_

std::unique_ptr<hpx::threads::threadmanager> thread_manager_
```

Protected Static Attributes

```
static 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_stacksizes)
```

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)
```

hpx/runtime_local/runtime_local_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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(hpx::function<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(hpx::function<void()> const&)`

Register a function to be called during system shutdown.

`bool is_starting()`

Test whether the runtime system is currently being started.

This function returns whether the runtime system is currently being started or not, e.g. whether the current state of the runtime system is *hpx::state::startup*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

`bool tolerate_node_faults()`

Test if HPX runs in fault-tolerant mode.

This function returns whether the runtime system is running in fault-tolerant mode

`bool is_running()`

Test whether the runtime system is currently running.

This function returns whether the runtime system is currently running or not, e.g. whether the current state of the runtime system is *hpx::state::running*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

`bool is_stopped()`

Test whether the runtime system is currently stopped.

This function returns whether the runtime system is currently stopped or not, e.g. whether the current state of the runtime system is *hpx::state::stopped*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

`bool is_stopped_or_shutting_down()`

Test whether the runtime system is currently being shut down.

This function returns whether the runtime system is currently being shut down or not, e.g. whether the current state of the runtime system is *hpx::state::stopped* or *hpx::state::shutdown*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

`std::size_t get_num_worker_threads()`

Return the number of worker OS- threads used to execute HPX threads.

This function returns the number of OS-threads used to execute HPX threads. If the function is called while no HPX runtime system is active, it will return zero.

`std::uint64_t get_system_uptime()`

Return the system uptime measure on the thread executing this call.

This function returns the system uptime measured in nanoseconds for the thread executing this call. If the function is called while no HPX runtime system is active, it will return zero.

namespace **threads**

hpx/runtime_local/service_executors.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Enums

enum class **service_executor_type**

Values:

enumerator **io_thread_pool**

Selects creating a service executor using the I/O pool of threads

enumerator **parcel_thread_pool**

Selects creating a service executor using the parcel pool of threads

enumerator **timer_thread_pool**

Selects creating a service executor using the timer pool of threads

enumerator **main_thread**

Selects creating a service executor using the main thread

struct **io_pool_executor** : public *service_executor*

Public Functions

inline **io_pool_executor()**

struct **main_pool_executor** : public *service_executor*

Public Functions

```
inline main_pool_executor()  
  
struct parcel_pool_executor : public service_executor
```

Public Functions

```
inline parcel_pool_executor(char const *name_suffix = "-tcp")  
  
struct service_executor : public service_executor
```

Public Functions

```
inline service_executor(service_executor_type t, char const *name_suffix = "")  
  
struct timer_pool_executor : public service_executor
```

Public Functions

```
inline timer_pool_executor()
```

hpx/runtime_local/shutdown_function.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Typedefs

```
typedef hpx::move_only_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()*).

See also:

`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.

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()`).

See also:

`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.

Parameters `f` – [in] The function to be registered to run by an HPX thread as a shutdown function.

hpx/runtime_local/startup_function.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace `hpx`

TypeDefs

`typedef hpx::move_only_function<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).

See also:

`hpx::register_startup_function()`

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.

```
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).

See also:

`hpx::register_pre_startup_function()`

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.

hpx/runtime_local/thread_hooks.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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*).

Note: This function can be called before the HPX runtime is initialized.

Returns The currently installed error handler function.

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*).

Note: This function can be called before the HPX runtime is initialized.

Returns The currently installed error handler function.

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*).

Note: This function can be called before the HPX runtime is initialized.

Returns The currently installed error handler function.

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*).

Note: This function can be called before the HPX runtime is initialized.

Parameters **f** – The function to install as the new start handler.

Returns The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

```
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*).

Note: This function can be called before the HPX runtime is initialized.

Parameters **f** – The function to install as the new stop handler.

Returns The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

```
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*).

Note: This function can be called before the HPX runtime is initialized.

Parameters **f** – The function to install as the new error handler.

Returns The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

hpx/runtime_local/thread_pool_helpers.hpp

See *Public API* for a list of names and headers that are part of the public *HPX* API.

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(hpx::function<bool(thread_id_type)> const &f, thread_schedule_state state = thread_schedule_state::unknown)`

The function `enumerate_threads` will invoke the given function `f` for each thread with a matching thread state.

Parameters

- **f** – [in] The function which should be called for each matching thread. Returning ‘false’ from this function will stop the enumeration process.
- **state** – [in] This specifies the thread-state for which the threads should be enumerated.

serialization

See [Public API](#) for a list of names and headers that are part of the public HPX API.

`hpx/serialization/base_object.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
template<typename Derived, typename Base>
struct hpx::serialization::base_object_type<Derived, Base, std::true_type>
```

Public Functions

```
inline explicit constexpr base_object_type(Derived &d) noexcept
```

```
template<typename Archive>
```

```
inline void save(Archive &ar, unsigned) const
```

```
template<typename Archive>
```

```
inline void load(Archive &ar, unsigned)
```

```
HPX_SERIALIZATION_SPLIT_MEMBER()
```

Public Members

Derived &d_

namespace **hpx**

namespace **serialization**

Functions

```
template<typename Base, typename Derived>
constexpr base_object_type<Derived, Base> base_object(Derived &d) noexcept

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>::type>
struct base_object_type
```

Public Functions

```
inline explicit constexpr base_object_type(Derived &d) noexcept

template<typename Archive>
inline void serialize(Archive &ar, unsigned)
```

Public Members

Derived &d_

```
template<typename Derived, typename Base> true_type >
```

Public Functions

```
inline explicit constexpr base_object_type(Derived &d) noexcept
template<typename Archive>
inline void save(Archive &ar, unsigned) const
template<typename Archive>
inline void load(Archive &ar, unsigned)
HPX_SERIALIZATION_SPLIT_MEMBER()
```

Public Members

Derived **&d_**

synchronization

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/synchronization/barrier.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

```
template<typename OnCompletion = detail::empty_oncompletion>
class barrier
#include <barrier.hpp> A barrier is a thread coordination mechanism whose lifetime consists of a sequence of barrier phases, where each phase allows at most an expected number of threads to block until the expected number of threads arrive at the barrier. [ Note: A barrier is useful for managing repeated tasks that are handled by multiple threads. - end note ] Each barrier phase consists of the following steps:
```

- The expected count is decremented by each call to `arrive` or `arrive_and_drop`.
- When the expected count reaches zero, the phase completion step is run. For the specialization with the default value of the `CompletionFunction` template parameter, the completion step is run as part of the call to `arrive` or `arrive_and_drop` that caused the expected count to reach zero. For other specializations, the completion step is run on one of the threads that arrived at the barrier during the phase.
- When the completion step finishes, the expected count is reset to what was specified by the `expected` argument to the constructor, possibly adjusted by calls to `arrive_and_drop`, and the next phase starts.

Each phase defines a phase synchronization point. Threads that arrive at the barrier during the phase can block on the phase synchronization point by calling `wait`, and will remain blocked until the phase completion step is run. The phase completion step that is executed at the end of each phase has the following effects:

- Invokes the completion function, equivalent to `completion()`.
- Unblocks all threads that are blocked on the phase synchronization point.

The end of the completion step strongly happens before the returns from all calls that were unblocked by the completion step. For specializations that do not have the default value of the CompletionFunction template parameter, the behavior is undefined if any of the barrier object's member functions other than wait are called while the completion step is in progress.

Concurrent invocations of the member functions of barrier, other than its destructor, do not introduce data races. The member functions arrive and arrive_and_drop execute atomically.

CompletionFunction shall meet the Cpp17MoveConstructible (Table 28) and Cpp17Destructible (Table 32) requirements. std::is_nothrow_invocable_v<CompletionFunction&> shall be true.

The default value of the CompletionFunction template parameter is an unspecified type, such that, in addition to satisfying the requirements of CompletionFunction, it meets the Cpp17DefaultConstructible requirements (Table 27) and completion() has no effects.

barrier::arrival_token is an unspecified type, such that it meets the Cpp17MoveConstructible (Table 28), Cpp17MoveAssignable (Table 30), and Cpp17Destructible (Table 32) requirements.

Public Types

```
using arrival_token = bool
```

Public Functions

inline explicit constexpr barrier(std::ptrdiff_t expected, *OnCompletion* completion = *OnCompletion*())

Preconditions: expected >= 0 is true and expected <= max() is true.

Effects: Sets both the initial expected count for each barrier phase and the current expected count for the first phase to expected. Initializes completion with std::move(f). Starts the first phase. [Note: If expected is 0 this object can only be destroyed.- end note]

Throws

inline *arrival_token* arrive(std::ptrdiff_t update = 1)

Preconditions: update > 0 is true, and update is less than or equal to the expected count for the current barrier phase.

Effects: Constructs an object of type arrival_token that is associated with the phase synchronization point for the current phase. Then, decrements the expected count by update.

Synchronization: The call to arrive strongly happens before the start of the phase completion step for the current phase.

Throws

Returns : The constructed arrival_token object.

inline void wait(*arrival_token* &&old_phase) const

Preconditions: arrival is associated with the phase synchronization point for the current phase or the immediately preceding phase of the same barrier object.

Effects: Blocks at the synchronization point associated with HPX_MOVE(arrival) until the phase completion step of the synchronization point's phase is run. [Note: If arrival is associated with the synchronization point for a previous phase, the call returns immediately. - end note]

Throws

inline void arrive_and_wait()

Effects: Equivalent to: wait(arrive()).

```
inline void arrive_and_drop()
```

Preconditions: The expected count for the current barrier phase is greater than zero.

Effects: Decrements the initial expected count for all subsequent phases by one. Then decrements the expected count for the current phase by one.

Synchronization: The call to `arrive_and_drop` strongly happens before the start of the phase completion step for the current phase.

Throws

Public Static Functions

```
static inline constexpr std::ptrdiff_t() max() noexcept
```

Private Types

```
using mutex_type = hpx::spinlock
```

Private Members

```
mutable mutex_type mtx_
```

```
mutable hpx::lcos::local::detail::condition_variable cond_
```

```
std::ptrdiff_t expected_
```

```
std::ptrdiff_t arrived_
```

```
OnCompletion completion_
```

```
bool phase_
```

hpx/synchronization/binary_semaphore.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

```
class binary_semaphore
```

```
#include <binary_semaphore.hpp> A binary semaphore is a semaphore object that has only two states. binary_semaphore is an alias for specialization of hpx::counting_semaphore with Least.MaxValue being 1. HPX's implementation of binary_semaphore is more efficient than the default implementation of a counting semaphore with a unit resource count (hpx::counting_semaphore).
```

Public Functions

```
binary_semaphore(binary_semaphore const&) = delete  
binary_semaphore &operator=(binary_semaphore const&) = delete  
binary_semaphore(binary_semaphore&&) = delete  
binary_semaphore &operator=(binary_semaphore&&) = delete  
explicit binary_semaphore(std::ptrdiff_t value = 1)
```

Constructs an object of type `hpx::binary_semaphore` with the internal counter initialized to `value`.

Parameters `value` – The initial value of the internal semaphore lock count. Normally this value should be zero (which is the default), values greater than zero are equivalent to the same number of signals pre-set, and negative values are equivalent to the same number of waits pre-set.

```
~binary_semaphore() = default
```

```
void release(std::ptrdiff_t update = 1)
```

Atomically increments the internal counter by the value of `update`. Any thread(s) waiting for the counter to be greater than 0, such as due to being blocked in `acquire`, will subsequently be unblocked.

Note: Synchronization: Strongly happens before invocations of `try_acquire` that observe the result of the effects.

Throws `std::system_error` –

Parameters `update` – the amount to increment the internal counter by

Pre Both `update >= 0` and `update <= max()` – `counter` are `true`, where `counter` is the value of the internal counter.

```
bool try_acquire() noexcept
```

Tries to atomically decrement the internal counter by 1 if it is greater than 0; no blocking occurs regardless.

Returns `true` if it decremented the internal counter, otherwise `false`

```
void acquire()
```

Repeatedly performs the following steps, in order:

- Evaluates `try_acquire`. If the result is true, returns.

Blocks on `*this` until counter is greater than zero.

Throws `std::system_error` –

Returns `void`.

```
bool try_acquire_until(hpx::chrono::steady_time_point const &abs_time)
```

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the `abs_time` time point has been passed.

Parameters `abs_time` – the earliest time the function must wait until in order to fail

Throws `std::system_error` –

Returns `true` if it decremented the internal counter, otherwise `false`.

`bool try_acquire_for(hpx::chrono::steady_duration const &rel_time)`

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the `rel_time` duration has been exceeded.

Throws `std::system_error` –

Parameters `rel_time` – the minimum duration the function must wait for to fail

Returns `true` if it decremented the internal counter, otherwise `false`

Public Static Functions

`static constexpr std::ptrdiff_t max() noexcept`

Returns The maximum value of counter. This value is greater than or equal to *Least.MaxValue*.

Returns The internal counter's maximum possible value, as a `std::ptrdiff_t`.

[hpx/synchronization/condition_variable.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Enums

enum class **cv_status**

The scoped enumeration `hpx::cv_status` describes whether a timed wait returned because of timeout or not. `hpx::cv_status` is used by the `wait_for` and `wait_until` member functions of `hpx::condition_variable` and `hpx::condition_variable_any`.

Values:

enumerator **no_timeout**

the condition variable was awakened with `notify_all`, `notify_one`, or spuriously

enumerator **timeout**

the condition variable was awakened by timeout expiration

enumerator **error**

there was an error

class **condition_variable**

`#include <condition_variable.hpp>` The `condition_variable` class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the condition), and notifies the `condition_variable`.

The thread that intends to modify the shared variable has to

- i. acquire a `hpx::mutex` (typically via `std::lock_guard`)
- ii. perform the modification while the lock is held

- iii. execute `notify_one` or `notify_all` on the `condition_variable` (the lock does not need to be held for notification)

Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread. Any thread that intends to wait on `condition_variable` has to

- i. acquire a `std::unique_lock<hpx::mutex>`, on the same mutex as used to protect the shared variable
- ii. either
 - A. check the condition, in case it was already updated and notified
 - B. execute `wait`, `await_for`, or `wait_until`. The wait operations atomically release the mutex and suspend the execution of the thread.
 - C. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious. or
 - A. use the predicated overload of `wait`, `wait_for`, and `wait_until`, which takes care of the three steps above.

`hpx::condition_variable` works only with `std::unique_lock<hpx::mutex>`. This restriction allows for maximal efficiency on some platforms. `hpx::condition_variable_any` provides a condition variable that works with any `BasicLockable`⁴⁸⁶ object, such as `std::shared_lock`.

Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

The class `hpx::condition_variable` is a `StandardLayoutType`⁴⁸⁷. It is not `CopyConstructible`⁴⁸⁸, `MoveConstructible`⁴⁸⁹, `CopyAssignable`⁴⁹⁰, or `MoveAssignable`⁴⁹¹.

Public Functions

inline **condition_variable()**

Construct an object of type `hpx::condition_variable`.

~condition_variable() = default

Destroys the object of type `hpx::condition_variable`.

IOW, `~condition_variable()` can execute before a signaled thread returns from a wait. If this happens with `condition_variable`, that waiting thread will attempt to lock the destructed mutex. To fix this, there must be shared ownership of the data members between the `condition_variable` object and the member functions `wait` (`wait_for`, etc.).

Note: Preconditions: There is no thread blocked on `*this`. [Note: That is, all threads have been notified; they could subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. The user should take care to ensure that no threads wait on `*this` once the destructor has been started, especially when the waiting threads are calling the `wait` functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. end note]

inline void **notify_one**(*error_code* &ec = *throws*)

If any threads are waiting on **this*, calling *notify_one* unblocks one of the waiting threads.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws*; A special ‘throw on error’ *error_code*.

Returns *notify_one* returns *void*.

inline void **notify_all**(*error_code* &ec = *throws*)

Unblocks all threads currently waiting for **this*.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws*; A special ‘throw on error’ *error_code*.

Returns *notify_all* returns *void*.

template<typename **Mutex**>

inline void **wait**(*std::unique_lock<Mutex>* &lock, *error_code* &ec = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (*bool(pred())==true*).

Atomically unlocks lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when *notify_all()* or *notify_one()* is executed. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and wait exits.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters **Mutex** – Type of mutex to wait on.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws*; A special ‘throw on error’ *error_code*.

Returns *wait* returns *void*.

template<typename **Mutex**, typename **Predicate**>

inline void **wait**(*std::unique_lock<Mutex>* &lock, *Predicate* pred, *error_code*& = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (*bool(pred())==true*).

Equivalent to

```
while (!pred()) {
    wait(lock);
}
```

This overload may be used to ignore spurious awakenings while waiting for a specific condition to become true. Note that lock must be acquired before entering this method, and it is reacquired after *wait(lock)* exits, which means that lock can be used to guard access to *pred()*.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters

- **Mutex** – Type of mutex to wait on.
- **Predicate** – Type of predicate *pred* function.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **pred** – Predicate which returns *false* if the waiting should be continued (*bool(pred())==false*). The signature of the predicate function should be equivalent to the following: *bool pred();*

Returns *wait* returns *void*.

```
template<typename Mutex>
inline cv_status wait_until(std::unique_lock<Mutex> &lock, hpx::chrono::steady_time_point const
                           &abs_time, error_code &ec = throws)
```

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (*bool(pred())==true*).

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when *notify_all()* or *notify_one()* is executed, or when the absolute time point *abs_time* is reached. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and *wait_until* exits.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.
A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters **Mutex** – Type of mutex to wait on.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **abs_time** – Represents the time when waiting should be stopped
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *cv_status* *wait_until* returns *hpx::cv_status::timeout* if the absolute timeout specified by *abs_time* was reached and *hpx::cv_status::no_timeout* otherwise.

```
template<typename Mutex, typename Predicate>
inline bool wait_until(std::unique_lock<Mutex> &lock, hpx::chrono::steady_time_point const
                      &abs_time, Predicate pred, error_code &ec = throws)
```

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (*bool(pred())==true*).

Equivalent to

```
while (!pred()) {
    if (wait_until(lock, abs_time) == hpx::cv_status::timeout) {
        return pred();
    }
}
return true;
```

This overload may be used to ignore spurious wakeups.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.
A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters

- **Mutex** – Type of mutex to wait on.
- **Predicate** – Type of predicate *pred* function.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **abs_time** – Represents the time when waiting should be stopped
- **pred** – Predicate which returns *false* if the waiting should be continued (*bool(pred())==false*). The signature of the predicate function should be equivalent to the following: *bool pred();*
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *bool wait_until* returns *false* if the predicate *pred* still evaluates to false after the *abs_time* timeout has expired, otherwise *true*. If the timeout had already expired, evaluates and returns the result of *pred*.

```
template<typename Mutex>
inline cv_status wait_for(std::unique_lock<Mutex> &lock, hpx::chrono::steady_duration const
&rel_time, error_code &ec = throws)
```

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when *notify_all()* or *notify_one()* is executed, or when the relative timeout *rel_time* expires. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and *wait_for()* exits.

The standard recommends that a steady clock be used to measure the duration. This function may block for longer than *rel_time* due to scheduling or resource contention delays.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.
A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.
B. Even if notified under lock, this overload makes no guarantees about the state of the associated predicate when returning due to timeout.

Template Parameters **Mutex** – Type of mutex to wait on.**Parameters**

- **lock** – *unique_lock* that must be locked by the current thread
- **rel_time** – represents the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *cv_status hpx::cv_status::timeout* if the relative timeout specified by *rel_time* expired, *hpx::cv_status::no_timeout* otherwise.

```
template<typename Mutex, typename Predicate>
inline bool wait_for(std::unique_lock<Mutex> &lock, hpx::chrono::steady_duration const &rel_time,
Predicate pred, error_code &ec = throws)
```

Equivalent to.

```
return wait_until(lock,
    hpx::chrono::steady_clock::now() + rel_time,
    hpx::move(pred));
```

This overload may be used to ignore spurious awakenings by looping until some predicate is satisfied (*bool(pred())==true*).

The standard recommends that a steady clock be used to measure the duration. This function may block for longer than *rel_time* due to scheduling or resource contention delays.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.
A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters

- **Mutex** – Type of mutex to wait on.
- **Predicate** – Type of predicate *pred* function.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **rel_time** – represents the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **pred** – Predicate which returns *false* if the waiting should be continued (*bool(pred())==false*). The signature of the predicate function should be equivalent to the following: *bool pred();*
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns bool *wait_for* returns *false* if the predicate *pred* still evaluates to *false* after the *rel_time* timeout expired, otherwise *true*.

Private Types

```
using mutex_type = lcos::local::detail::condition_variable_data::mutex_type
```

```
using data_type = hpx::intrusive_ptr<lcos::local::detail::condition_variable_data>
```

Private Members

```
hpx::util::cache_aligned_data_derived<data_type> data_
```

```
class condition_variable_any
```

```
#include <condition_variable.hpp> The condition_variable_any class is a generalization of hpx::condition_variable. Whereas hpx::condition_variable works only on std::unique_lock<std::mutex>, a condition_variable_any can operate on any lock that meets the BasicLockable492 requirements.
```

See [hpx::condition_variable](#) for the description of the semantics of condition variables. It is not [CopyConstructible](#)⁴⁹³, [MoveConstructible](#)⁴⁹⁴, [CopyAssignable](#)⁴⁹⁵, or [MoveAssignable](#)⁴⁹⁶.

Public Functions

inline `condition_variable_any()`

Constructs an object of type `hpx::condition_variable_any`.

`~condition_variable_any() = default`

Destroys the object of type `hpx::condition_variable_any`.

It is only safe to invoke the destructor if all threads have been notified. It is not required that they have exited their respective wait functions: some threads may still be waiting to reacquire the associated lock, or may be waiting to be scheduled to run after reacquiring it.

The programmer must ensure that no threads attempt to wait on `*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or are using the overloads of the wait functions that take a predicate.

Preconditions: There is no thread blocked on `*this`. [Note: That is, all threads have been notified; they could subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. The user should take care to ensure that no threads wait on `*this` once the destructor has been started, especially when the waiting threads are calling the `wait` functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. end note]

IOW, `~condition_variable_any()` can execute before a signaled thread returns from a wait. If this happens with `condition_variable_any`, that waiting thread will attempt to lock the destructed mutex. To fix this, there must be shared ownership of the data members between the `condition_variable_any` object and the member functions `wait` (`wait_for`, etc.).

inline void `notify_one(error_code &ec = throws)`

If any threads are waiting on `*this`, calling `notify_one` unblocks one of the waiting threads.

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock. However, some implementations (in particular many implementations of pthreads) recognize this situation and avoid this “hurry up and wait” scenario by transferring the waiting thread from the condition variable’s queue directly to the queue of the mutex within the notify call, without waking it up.

Notifying while under the lock may nevertheless be necessary when precise scheduling of events is required, e.g. if the waiting thread would exit the program if the condition is satisfied, causing destruction of the notifying thread’s condition variable. A spurious wakeup after mutex unlock but before notify would result in notify called on a destroyed object.

Note: The effects of `notify_one()`/`notify_all()` and each of the three atomic parts of `wait()`/`wait_for()`/`wait_until()` (unlock+wait wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for `notify_one()` to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Parameters `ec` – Used to hold error code value originated during the operation. Defaults to `throws`; A special ‘throw on error’ `error_code`.

Returns `notify_one` returns `void`.

```
inline void notify_all(error_code &ec = throws)
```

Unblocks all threads currently waiting for **this*.

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock.

Note: The effects of *notify_one()*/*notify_all()* and each of the three atomic parts of *wait()*/*wait_for()*/*wait_until()* (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to *notify_one()* was made.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *notify_all* returns *void*.

```
template<typename Lock>
```

```
inline void wait(Lock &lock, error_code &ec = throws)
```

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (*bool(pred())==true*).

Atomically unlocks *lock*, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when *notify_all()* or *notify_one()* is executed. It may also be unblocked spuriously. When unblocked, regardless of the reason, *lock* is reacquired and *wait* exits.

The effects of *notify_one()*/*notify_all()* and each of the three atomic parts of *wait()*/*wait_for()*/*wait_until()* (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to *notify_one()* was made.

Note: If these functions fail to meet the postconditions (lock is locked by the calling thread), *std::terminate* is called. For example, this could happen if relocking the mutex throws an exception.

Template Parameters **Lock** – Type of *lock*.

Parameters

- **lock** – An object of type *Lock* that meets the *BasicLockable*⁴⁹⁷ requirements, which must be locked by the current thread
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *wait* returns *void*.

```
template<typename Lock, typename Predicate>
```

```
inline void wait(Lock &lock, Predicate pred, error_code& = throws)
```

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (*bool(pred())==true*).

Equivalent to

```
while (!pred()) {
    wait(lock);
}
```

This overload may be used to ignore spurious awakenings while waiting for a specific condition to become true. Note that lock must be acquired before entering this method, and it is reacquired after *wait(lock)* exits, which means that lock can be used to guard access to *pred()*.

The effects of *notify_one()*/*notify_all()* and each of the three atomic parts of *wait()*/*wait_for()*/*wait_until()* (*unlock+wait*, *wakeup*, and *lock*) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to *notify_one()* was made.

Note: If these functions fail to meet the postconditions (lock is locked by the calling thread), `std::terminate` is called. For example, this could happen if relocking the mutex throws an exception.

Template Parameters

- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters

- **lock** – an object of type *Lock* that meets the `BasicLockable`⁴⁹⁸ requirements, which must be locked by the current thread
- **pred** – predicate which returns `false` if the waiting should be continued (`bool(pred()) == false`). The signature of the predicate function should be equivalent to the following: `bool pred()`.

Returns *wait* returns `void`.

```
template<typename Lock>
inline cv_status wait_until(Lock &lock, hpx::chrono::steady_time_point const &abs_time, error_code
                           &ec = throws)
```

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (`bool(pred()) == true`).

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when *notify_all()* or *notify_one()* is executed, or when the absolute time point *abs_time* is reached. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and *wait_until* exits.

Note: The effects of *notify_one()*/*notify_all()* and each of the three atomic parts of *wait()*/*wait_for()*/*wait_until()* (*unlock+wait*, *wakeup*, and *lock*) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to *notify_one()* was made.

Template Parameters **Lock** – Type of *lock*.

Parameters

- **lock** – an object of type *Lock* that meets the requirements of `BasicLockable`⁴⁹⁹, which must be locked by the current thread
- **abs_time** – represents the time when waiting should be stopped.

- **ec** – used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns `cv_status hpx::cv_status::timeout` if the absolute timeout specified by *abs_time* was reached, `hpx::cv_status::no_timeout` otherwise.

```
template<typename Lock, typename Predicate>
inline bool wait_until(Lock &lock, hpx::chrono::steady_time_point const &abs_time, Predicate pred,
                      error_code &ec = throws)
```

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (`bool(pred()) == true`).

Equivalent to

```
while (!pred()) {
    if (wait_until(lock, timeout_time) == hpx::cv_status::timeout) {
        return pred();
    }
}
return true;
```

This overload may be used to ignore spurious wakeups.

Note: The effects of `notify_one()`/`notify_all()` and each of the three atomic parts of `wait()`/`wait_for()`/`wait_until()` (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for `notify_one()` to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Template Parameters

- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters

- **lock** – an object of type *Lock* that meets the requirements of `BasicLockable`⁵⁰⁰, which must be locked by the current thread
- **abs_time** – represents the time when waiting should be stopped.
- **pred** – predicate which returns *false* if the waiting should be continued (`bool(pred()) == false`). The signature of the predicate function should be equivalent to the following: `bool pred();`
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns `bool false` if the predicate *pred* still evaluates to *false* after the *abs_time* timeout expired, otherwise `true`. If the timeout had already expired, evaluates and returns the result of *pred*.

```
template<typename Lock>
inline cv_status wait_for(Lock &lock, hpx::chrono::steady_duration const &rel_time, error_code &ec
                         = throws)
```

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when `notify_all()` or `notify_one()` is executed, or when the relative timeout *rel_time* expires. It may also be unblocked spuriously. When unblocked, regardless of the reason, *lock* is reacquired and `wait_for()` exits.

The effects of `notify_one()`/`notify_all()` and each of the three atomic parts of `wait()`/`wait_for()`/`wait_until()` (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for `notify_one()` to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Note: Even if notified under lock, this overload makes no guarantees about the state of the associated predicate when returning due to timeout.

Template Parameters **Lock** – Type of *lock*.

Parameters

- **lock** – an object of type *Lock* that meets the [BasicLockable](#)⁵⁰¹ requirements, which must be locked by the current thread.
- **rel_time** – an object of type *hpx::chrono::duration* representing the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns `cv_status hpx::cv_status::timeout` if the relative timeout specified by *rel_time* expired, `hpx::cv_status::no_timeout` otherwise.

template<typename **Lock**, typename **Predicate**>

inline bool **wait_for**(*Lock* &*lock*, *hpx::chrono::steady_duration* const &*rel_time*, *Predicate* *pred*, *error_code* &*ec* = *throws*)

Equivalent to.

```
return wait_until(lock,
    hpx::chrono::steady_clock::now() + rel_time,
    std::move(pred));
```

This overload may be used to ignore spurious awakenings by looping until some predicate is satisfied (`bool(pred()) == true`).

Note: The effects of `notify_one()`/`notify_all()` and each of the three atomic parts of `wait()`/`wait_for()`/`wait_until()` (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for `notify_one()` to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Template Parameters

- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters

- **lock** – an object of type *Lock* that meets the [BasicLockable](#)⁵⁰² requirements, which must be locked by the current thread.
- **rel_time** – an object of type *hpx::chrono::duration* representing the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **pred** – predicate which returns *false* if the waiting should be continued (`bool(pred()) == false`). The signature of the predicate function should be equivalent to the following: `bool pred();`

- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns bool *false* if the predicate *pred* still evaluates to *false* after the *rel_time* timeout expired, otherwise *true*.

template<typename **Lock**, typename **Predicate**>

inline bool **wait**(*Lock* &*lock*, *stop_token* *stoken*, *Predicate* *pred*, *error_code* &*ec* = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(*pred*())==true).

An interruptible wait: registers the *condition_variable_any* for the duration of *wait()*, to be notified if a stop request is made on the given stoken’s associated stop-state; it is then equivalent to

```
while (!stoken.stop_requested()) {  
    if (pred()) return true;  
    wait(lock);  
}  
return pred();
```

Note that the returned value indicates whether *pred* evaluated to *true*, regardless of whether there was a stop requested or not.

Note: The effects of *notify_one()*/*notify_all()* and each of the three atomic parts of *wait()*/*wait_for()*/*wait_until()* (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to *notify_one()* was made.

Template Parameters

- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters

- **lock** – an object of type *Lock* that meets the *BasicLockable*⁵⁰³ requirements, which must be locked by the current thread
- **stoken** – a *hpx::stop_token* to register interruption for
- **pred** – predicate which returns *false* if the waiting should be continued (bool(*pred*()) == *false*). The signature of the predicate function should be equivalent to the following: bool *pred*().
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns bool result of *pred*().

template<typename **Lock**, typename **Predicate**>

inline bool **wait_until**(*Lock* &*lock*, *stop_token* *stoken*, *hpx::chrono::steady_time_point* const &*abs_time*, *Predicate* *pred*, *error_code* &*ec* = *throws*)

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(*pred*()) == *true*).

An interruptible wait: registers the *condition_variable_any* for the duration of *wait_until()*, to be notified if a stop request is made on the given stoken’s associated stop-state; it is then equivalent to

```
while (!stoken.stop_requested()) {  
    if (pred())
```

(continues on next page)

(continued from previous page)

```

    return true;
if (wait_until(lock, timeout_time) == hpx::cv_status::timeout)
    return pred();
}
return pred();

```

Note: The effects of `notify_one()`/`notify_all()` and each of the three atomic parts of `wait()`/`wait_for()`/`wait_until()` (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for `notify_one()` to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Template Parameters

- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters

- **lock** – an object of type *Lock* that meets the requirements of `BasicLockable`⁵⁰⁴, which must be locked by the current thread.
- **stoken** – a `hpx::stop_token` to register interruption for.
- **abs_time** – represents the time when waiting should be stopped.
- **pred** – predicate which returns `false` if the waiting should be continued (`bool(pred()) == false`). The signature of the predicate function should be equivalent to the following: `bool pred();`.
- **ec** – Used to hold error code value originated during the operation. Defaults to `throws` — A special ‘throw on error’ `error_code`.

Returns `bool pred()`, regardless of whether the timeout was met or stop was requested.

```
template<typename Lock, typename Predicate>
inline bool wait_for(Lock &lock, stop_token stoken, hpx::chrono::steady_duration const &rel_time,
                     Predicate pred, error_code &ec = throws)
```

Equivalent to.

```

return wait_until(lock, std::move(stoken),
                  hpx::chrono::steady_clock::now() + rel_time,
                  std::move(pred));

```

Note: The effects of `notify_one()`/`notify_all()` and each of the three atomic parts of `wait()`/`wait_for()`/`wait_until()` (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for `notify_one()` to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Template Parameters

- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters

- **lock** – an object of type *Lock* that meets the `BasicLockable`⁵⁰⁵ requirements, which must be locked by the current thread.
- **stoken** – a `hpx::stop_token` to register interruption for.

- **rel_time** – an object of type *hpx::chrono::duration* representing the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **pred** – predicate which returns *false* if the waiting should be continued (*bool(pred()) == false*). The signature of the predicate function should be equivalent to the following: *bool pred();*
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *bool pred()*, regardless of whether the timeout was met or stop was requested.

Private Types

```
using mutex_type = lcos::local::detail::condition_variable_data::mutex_type
```

```
using data_type = hpx::intrusive_ptr<lcos::local::detail::condition_variable_data>
```

Private Members

```
hpx::util::cache_aligned_data_derived<data_type> data_
```

namespace **lcos**

namespace **local**

Typedefs

```
typedef hpx::condition_variable instead
```

⁴⁸⁶ https://en.cppreference.com/w/cpp/named_req/BasicLockable

⁴⁸⁷ https://en.cppreference.com/w/cpp/named_req/StandardLayoutType

⁴⁸⁸ https://en.cppreference.com/w/cpp/named_req/CopyConstructible

⁴⁸⁹ https://en.cppreference.com/w/cpp/named_req/MoveConstructible

⁴⁹⁰ https://en.cppreference.com/w/cpp/named_req/CopyAssignable

⁴⁹¹ https://en.cppreference.com/w/cpp/named_req/MoveAssignable

⁴⁹² https://en.cppreference.com/w/cpp/named_req/BasicLockable

⁴⁹³ https://en.cppreference.com/w/cpp/named_req/CopyConstructible

⁴⁹⁴ https://en.cppreference.com/w/cpp/named_req/MoveConstructible

495 https://en.cppreference.com/w/cpp/named_req/CopyAssignable496 https://en.cppreference.com/w/cpp/named_req/MoveAssignable497 https://en.cppreference.com/w/cpp/named_req/BasicLockable498 https://en.cppreference.com/w/cpp/named_req/BasicLockable499 https://en.cppreference.com/w/cpp/named_req/BasicLockable500 https://en.cppreference.com/w/cpp/named_req/BasicLockable501 https://en.cppreference.com/w/cpp/named_req/BasicLockable502 https://en.cppreference.com/w/cpp/named_req/BasicLockable503 https://en.cppreference.com/w/cpp/named_req/BasicLockable504 https://en.cppreference.com/w/cpp/named_req/BasicLockable505 https://en.cppreference.com/w/cpp/named_req/BasicLockable

hpx/synchronization/counting_semaphore.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

```
template<std::ptrdiff_t Least.MaxValue = PTRDIFF_MAX>
```

```
class counting_semaphore
```

#include <counting_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.

Counting 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. Unlike [hpx::mutex](#) a [counting_semaphore](#) is not tied to threads of execution; acquiring a semaphore can occur on a different thread than releasing the semaphore, for example. All operations on [counting_semaphore](#) can be performed concurrently and without any relation to specific threads of execution, with the exception of the destructor which cannot be performed concurrently but can be performed on a different thread.

Semaphores are lightweight synchronization primitives used to constrain concurrent access to a shared resource. They are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.

A counting semaphore is a semaphore object that models a non-negative resource count.

Class template [counting_semaphore](#) maintains an internal counter that is initialized when the semaphore is created. The counter is decremented when a thread acquires the semaphore, and is incremented when a thread releases the semaphore. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

Specializations of [hpx::counting_semaphore](#) are not [DefaultConstructible](#)⁵⁰⁶, [CopyConstructible](#)⁵⁰⁷, [MoveConstructible](#)⁵⁰⁸, [CopyAssignable](#)⁵⁰⁹, or [MoveAssignable](#)⁵¹⁰.

Note: *couting_semaphore*'s [try_acquire\(\)](#) can spuriously fail.

Template Parameters `Least.MaxValue` – [counting_semaphore](#) allows more than one concurrent access to the same resource, for at least *Least.MaxValue* concurrent accessors. As its name indicates, the *Least.MaxValue* is the minimum max value, not the actual max value. Thus [max\(\)](#) can yield a number larger than *Least.MaxValue*.

Public Functions

```
counting_semaphore(counting_semaphore const&) = delete  
counting_semaphore &operator=(counting_semaphore const&) = delete  
counting_semaphore(counting_semaphore&&) = delete  
counting_semaphore &operator=(counting_semaphore&&) = delete  
explicit counting_semaphore(std::ptrdiff_t value)
```

Constructs an object of type `hpx::counting_semaphore` with the internal counter initialized to *value*.

Parameters **value** – The initial value of the internal semaphore lock count. Normally this value should be zero (which is the default), values greater than zero are equivalent to the same number of signals pre-set, and negative values are equivalent to the same number of waits pre-set.

```
~counting_semaphore() = default
```

```
void release(std::ptrdiff_t update = 1)
```

Atomically increments the internal counter by the value of *update*. Any thread(s) waiting for the counter to be greater than 0, such as due to being blocked in `acquire`, will subsequently be unblocked.

Note: Synchronization: Strongly happens before invocations of `try_acquire` that observe the result of the effects.

Throws `std::system_error` –

Parameters **update** – the amount to increment the internal counter by

Pre Both `update >= 0` and `update <= max()` – `counter` are *true*, where `counter` is the value of the internal counter.

```
bool try_acquire() noexcept
```

Tries to atomically decrement the internal counter by 1 if it is greater than 0; no blocking occurs regardless.

Returns *true* if it decremented the internal counter, otherwise *false*

```
void acquire()
```

Repeatedly performs the following steps, in order:

- Evaluates `try_acquire`. If the result is true, returns.
- Blocks on `*this` until counter is greater than zero.

Throws `std::system_error` –

Returns *void*.

```
bool try_acquire_until(hpx::chrono::steady_time_point const &abs_time)
```

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the `abs_time` time point has been passed.

Parameters **abs_time** – the earliest time the function must wait until in order to fail

Throws `std::system_error` –

Returns *true* if it decremented the internal counter, otherwise *false*.

```
bool try_acquire_for(hpx::chrono::steady_duration const &rel_time)
```

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the *rel_time* duration has been exceeded.

Throws *std*::system_error –

Parameters **rel_time** – the minimum duration the function must wait for to fail

Returns *true* if it decremented the internal counter, otherwise false

Public Static Functions

```
static constexpr std::ptrdiff_t max() noexcept
```

Returns The maximum value of counter. This value is greater than or equal to *Least.MaxValue*.

Returns The internal counter's maximum possible value, as a *std::ptrdiff_t*.

```
template<typename Mutex = hpx::spinlock, int N = 0>
```

```
class counting_semaphore_var
```

#include <counting_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.

Counting 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. Unlike *hpx::mutex* a *counting_semaphore_var* is not tied to threads of execution — acquiring a semaphore can occur on a different thread than releasing the semaphore, for example. All operations on *counting_semaphore_var* can be performed concurrently and without any relation to specific threads of execution, with the exception of the destructor which cannot be performed concurrently but can be performed on a different thread.

Semaphores are lightweight synchronization primitives used to constrain concurrent access to a shared resource. They are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.

A counting semaphore is a semaphore object that models a non-negative resource count.

Class template *counting_semaphore_var* maintains an internal counter that is initialized when the semaphore is created. The counter is decremented when a thread acquires the semaphore, and is incremented when a thread releases the semaphore. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

Specializations of *hpx::counting_semaphore_var* are not DefaultConstructible⁵¹¹, CopyConstructible⁵¹², MoveConstructible⁵¹³, CopyAssignable⁵¹⁴, or MoveAssignable⁵¹⁵.

Note: *counting_semaphore_var*'s *try_acquire()* can spuriously fail.

Template Parameters

- **Mutex** – Type of mutex
- **N** – The initial value of the internal semaphore lock count.

Public Functions

explicit **counting_semaphore_var**(*std*::ptrdiff_t value = *N*)

Constructs an object of type *hpx::counting_semaphore_value* with the internal counter initialized to *N*.

Parameters **value** – The initial value of the internal semaphore lock count. Normally this value should be zero, values greater than zero are equivalent to the same number of signals pre-set, and negative values are equivalent to the same number of waits pre-set. Defaults to *N* (which in turn defaults to zero).

counting_semaphore_var(*counting_semaphore_var* const&) = delete

counting_semaphore_var &**operator=**(*counting_semaphore_var* const&) = delete

void **wait**(*std*::ptrdiff_t count = 1)

Wait for the semaphore to be signaled.

Parameters **count** – The value by which the internal lock count will be decremented. At the same time this is the minimum value of the lock count at which the thread is not yielded.

bool **try_wait**(*std*::ptrdiff_t count = 1)

Try to wait for the semaphore to be signaled.

Parameters **count** – The value by which the internal lock count will be decremented. At the same time this is the minimum value of the lock count at which the thread is not yielded.

Returns *try_wait* returns true if the calling thread was able to acquire the requested amount of credits. *try_wait* returns false if not sufficient credits are available at this point in time.

void **signal**(*std*::ptrdiff_t count = 1)

Signal the semaphore.

Parameters **count** – The value by which the internal lock count will be incremented.

std::ptrdiff_t **signal_all**()

Unblock all acquirers.

Returns *std*::ptrdiff_t internal lock count after the operation.

void **release**(*std*::ptrdiff_t update = 1)

Atomically increments the internal counter by the value of update. Any thread(s) waiting for the counter to be greater than 0, such as due to being blocked in acquire, will subsequently be unblocked.

Note: Synchronization: Strongly happens before invocations of *try_acquire* that observe the result of the effects.

Throws *std*::system_error –

Parameters **update** – the amount to increment the internal counter by

Pre Both *update* ≥ 0 and *update* $\leq \text{max}()$ – *counter* are *true*, where *counter* is the value of the internal counter.

bool **try_acquire**() noexcept

Tries to atomically decrement the internal counter by 1 if it is greater than 0; no blocking occurs regardless.

Returns *true* if it decremented the internal counter, otherwise *false*

void **acquire**()

Repeatedly performs the following steps, in order:

- Evaluates *try_acquire*. If the result is true, returns.

Blocks on `*this` until counter is greater than zero.

Throws `std::system_error` –
Returns `void`.

`bool try_acquire_until(hpx::chrono::steady_time_point const &abs_time)`

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the `abs_time` time point has been passed.

Parameters `abs_time` – the earliest time the function must wait until in order to fail
Throws `std::system_error` –
Returns `true` if it decremented the internal counter, otherwise `false`.

`bool try_acquire_for(hpx::chrono::steady_duration const &rel_time)`

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the `rel_time` duration has been exceeded.

Throws `std::system_error` –
Parameters `rel_time` – the minimum duration the function must wait for to fail
Returns `true` if it decremented the internal counter, otherwise false

Public Static Functions

`static constexpr std::ptrdiff_t max()` noexcept

Returns The maximum value of counter. This value is greater than or equal to *Least.MaxValue*.

Returns The internal counter's maximum possible value, as a `std::ptrdiff_t`.

Private Types

`using mutex_type = Mutex`

[hpx/synchronization/event.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace `hpx`

namespace `lcos`

namespace `local`

⁵⁰⁶ https://en.cppreference.com/w/cpp/named_req/DefaultConstructible
⁵⁰⁷ https://en.cppreference.com/w/cpp/named_req/CopyConstructible
⁵⁰⁸ https://en.cppreference.com/w/cpp/named_req/MoveConstructible
⁵⁰⁹ https://en.cppreference.com/w/cpp/named_req/CopyAssignable
⁵¹⁰ https://en.cppreference.com/w/cpp/named_req/MoveAssignable
⁵¹¹ https://en.cppreference.com/w/cpp/named_req/DefaultConstructible
⁵¹² https://en.cppreference.com/w/cpp/named_req/CopyConstructible
⁵¹³ https://en.cppreference.com/w/cpp/named_req/MoveConstructible
⁵¹⁴ https://en.cppreference.com/w/cpp/named_req/CopyAssignable
⁵¹⁵ https://en.cppreference.com/w/cpp/named_req/MoveAssignable

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**inline event()**

Construct a new event semaphore.

inline bool occurred()

Check if the event has occurred.

inline void wait()

Wait for the event to occur.

inline void set()

Release all threads waiting on this semaphore.

inline void reset()

Reset the event.

Private Types

typedef *hpx::spinlock mutex_type*

Private Functions

inline void wait_locked(*std::unique_lock<mutex_type> &l*)

inline 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_

hpx/synchronization/latch.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

class **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(latch)

inline explicit **latch**([std](#)::ptrdiff_t count)

Initialize the latch

Requires: count ≥ 0 . Synchronization: None Postconditions: counter_ == count.

~latch() = default

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.

inline void **count_down**([std](#)::ptrdiff_t update)

Decrement counter_ by n. Does not block.

Requires: counter_ $\geq n$ and $n \geq 0$.

Synchronization: Synchronizes with all calls that block on this latch and with all [try_wait](#) calls on this latch that return true .

Throws Nothing. –

inline bool **try_wait()** const noexcept

Returns: With very low probability false. Otherwise counter == 0.

inline void **wait()** const

If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization point until counter_ reaches 0.

Throws Nothing. –

```
inline void arrive_and_wait(std::ptrdiff_t update = 1)  
Effects: Equivalent to: count_down(update); wait();
```

Public Static Functions

```
static inline constexpr std::ptrdiff_t() max () noexcept
```

Returns: The maximum value of counter that the implementation supports.

Protected Types

```
using mutex_type = hpx::spinlock
```

Protected Attributes

```
mutable util::cache_line_data<mutex_type> mtx_
```

```
mutable util::cache_line_data<hpx::lcos::local::detail::condition_variable> cond_
```

```
std::atomic<std::ptrdiff_t> counter_
```

```
bool notified_
```

```
namespace lcos
```

```
namespace local
```

```
class latch : public hpx::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 *hpx::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 *hpx::distributed::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*)

inline explicit **latch**(*std*::ptrdiff_t count)

Initialize the latch

Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

~latch() = default

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.

inline 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.

Throws Nothing. –

inline bool **is_ready()** const noexcept

Returns: counter_ == 0. Does not block.

Throws Nothing. –

inline void **abort_all()**

inline void **count_up**(*std*::ptrdiff_t n)

Increments counter_ by n. Does not block.

Requires: n >= 0.

Throws Nothing. –

inline void **reset**(*std*::ptrdiff_t n)

Reset counter_ to n. Does not block.

Requires: n >= 0.

Throws Nothing. –

inline 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

hpx/synchronization/mutex.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

class **mutex**

`#include <mutex.hpp>` *mutex* class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. *mutex* offers exclusive, non-recursive ownership semantics:

- A calling thread owns a *mutex* from the time that it successfully calls either *lock* or *try_lock* until it calls *unlock*.
- When a thread owns a *mutex*, all other threads will block (for calls to *lock*) or receive a *false* return value (for *try_lock*) if they attempt to claim ownership of the *mutex*.
- A calling thread must not own the *mutex* prior to calling *lock* or *try_lock*.

The behavior of a program is undefined if a *mutex* is destroyed while still owned by any threads, or a thread terminates while owning a *mutex*. The *mutex* class satisfies all requirements of [Mutex⁵¹⁶](#) and [StandardLayoutType⁵¹⁷](#).

`hpx::mutex` is neither copyable nor movable.

Subclassed by `hpx::timed_mutex`

Public Functions

HPX_NON_COPYABLE(*mutex*)

`hpx::mutex` is neither copyable nor movable

mutex(char const *const description = "")

Constructs the *mutex*. The *mutex* is in unlocked state after the constructor completes.

Note: Because the default constructor is *constexpr*, static mutexes are initialized as part of static non-local initialization, before any dynamic non-local initialization begins. This makes it safe to lock a *mutex* in a constructor of any static object.

Parameters **description** – description of the *mutex*.

~mutex()

Destroys the *mutex*. The behavior is undefined if the *mutex* is owned by any thread or if any thread terminates while holding any ownership of the *mutex*.

void lock(char const *description, *error_code* &ec = *throws*)

Locks the *mutex*. If another thread has already locked the *mutex*, a call to *lock* will block execution until the lock is acquired. If *lock* is called by a thread that already owns the *mutex*, the behavior is undefined: for example, the program may deadlock. `hpx::mutex` can detect the invalid usage and throws a `std::system_error` with error condition `resource_deadlock_would_occur` instead of deadlock-ing. Prior *unlock()* operations on the same *mutex* synchronize- with (as defined in `std::memory_order`) this operation.

Note: `lock()` is usually not called directly: `std::unique_lock`, `std::scoped_lock`, and `std::lock_guard` are used to manage exclusive locking.

Parameters

- **description** – Description of the `mutex`
- **ec** – Used to hold error code value originated during the operation. Defaults to `throws` — A special ‘throw on error’ `error_code`.

Returns `void lock` returns `void`.

inline void **lock**(`error_code` &`ec` = `throws`)

Locks the mutex. If another thread has already locked the mutex, a call to lock will block execution until the lock is acquired. If lock is called by a thread that already owns the mutex, the behavior is undefined: for example, the program may deadlock. `hpx::mutex` can detect the invalid usage and throws a `std::system_error` with error condition `resource_deadlock_would_occur` instead of deadlock-ing. Prior `unlock()` operations on the same mutex synchronize - with(as defined in `std::memory_order`) this operation.

Note: `lock()` is usually not called directly: `std::unique_lock`, `std::scoped_lock`, and `std::lock_guard` are used to manage exclusive locking. This overload essentially calls `void lock(char const* description, error_code& ec = throws);` with `description` as `mutex::lock`.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to `throws` — A special ‘throw on error’ `error_code`.

Returns `void lock` returns `void`.

bool **try_lock**(`char const *description, error_code` &`ec` = `throws`)

Tries to lock the `mutex`. Returns immediately. On successful lock acquisition returns `true`, otherwise returns `false`. This function is allowed to fail spuriously and return `false` even if the `mutex` is not currently locked by any other thread. If `try_lock` is called by a thread that already owns the `mutex`, the behavior is undefined. Prior `unlock()` operation on the same mutex synchronizes-with (as defined in `std::memory_order`) this operation if it returns `true`. Note that prior `lock()` does not synchronize with this operation if it returns `false`.

Parameters

- **description** – Description of the `mutex`
- **ec** – Used to hold error code value originated during the operation. Defaults to `throws` — A special ‘throw on error’ `error_code`.

Returns `bool try_lock` returns `true` on successful lock acquisition, otherwise returns `false`.

inline bool **try_lock**(`error_code` &`ec` = `throws`)

Tries to lock the `mutex`. Returns immediately. On successful lock acquisition returns `true`, otherwise returns `false`. This function is allowed to fail spuriously and return `false` even if the `mutex` is not currently locked by any other thread. If `try_lock` is called by a thread that already owns the `mutex`, the behavior is undefined. Prior `unlock()` operation on the same mutex synchronizes-with (as defined in `std::memory_order`) this operation if it returns `true`. Note that prior `lock()` does not synchronize with this operation if it returns `false`.

Note: This overload essentially calls

```
void try_lock(char const* description,
             error_code& ec = throws);
```

with *description* as `mutex::try_lock`.

Parameters `ec` – Used to hold error code value originated during the operation. Defaults to `throws`; A special ‘throw on error’ `error_code`.

Returns `bool try_lock` returns `true` on successful lock acquisition, otherwise returns `false`.

`void unlock(error_code &ec = throws)`

Unlocks the *mutex*. The *mutex* must be locked by the current thread of execution, otherwise, the behavior is undefined. This operation *synchronizes-with* (as defined in `std::memory_order`) any subsequent *lock* operation that obtains ownership of the same *mutex*.

Parameters `ec` – Used to hold error code value originated during the operation. Defaults to `throws`; A special ‘throw on error’ `error_code`.

Returns `unlock` returns `void`.

`class timed_mutex : private hpx::mutex`

`#include <mutex.hpp>` The `timed_mutex` class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In a manner similar to `mutex`, `timed_mutex` offers exclusive, non-recursive ownership semantics. In addition, `timed_mutex` provides the ability to attempt to claim ownership of a `timed_mutex` with a timeout via the member functions `try_lock_for()` and `try_lock_until()`. The `timed_mutex` class satisfies all requirements of `TimedMutex`⁵¹⁸ and `StandardLayoutType`⁵¹⁹.

`hpx::timed_mutex` is neither copyable nor movable.

Public Functions

`HPX_NON_COPYABLE(timed_mutex)`

`hpx::timed_mutex` is neither copyable nor movable

`timed_mutex(char const *const description = "")`

Constructs a `timed_mutex`. The mutex is in unlocked state after the call.

Parameters `description` – Description of the `timed_mutex`.

`~timed_mutex()`

Destroys the `timed_mutex`. The behavior is undefined if the mutex is owned by any thread or if any thread terminates while holding any ownership of the mutex.

`bool try_lock_until(hpx::chrono::steady_time_point const &abs_time, char const *description, error_code &ec = throws)`

Tries to lock the mutex. Blocks until specified `abs_time` has been reached or the lock is acquired, whichever comes first. On successful lock acquisition returns `true`, otherwise returns `false`. If `abs_time` has already passed, this function behaves like `try_lock()`. As with `try_lock()`, this function is allowed to fail spuriously and return `false` even if the mutex was not locked by any other thread at some point before `abs_time`. Prior `unlock()` operation on the same mutex *synchronizes-with* (as defined in `std::memory_order`) this operation if it returns `true`. If `try_lock_until` is called by a thread that already owns the mutex, the behavior is undefined.

Parameters

- `abs_time` – time point to block until
- `description` – Description of the `timed_mutex`
- `ec` – Used to hold error code value originated during the operation. Defaults to `throws`; A special ‘throw on error’ `error_code`.

Returns `bool try_lock_until` returns `true` if the lock was acquired successfully, otherwise `false`.

```
inline bool try_lock_until(hpx::chrono::steady_time_point const &abs_time, error_code &ec = throws)
```

Tries to lock the mutex. Blocks until specified *abs_time* has been reached or the lock is acquired, whichever comes first. On successful lock acquisition returns *true*, otherwise returns *false*. If *abs_time* has already passed, this function behaves like *try_lock()*. As with *try_lock()*, this function is allowed to fail spuriously and return *false* even if the mutex was not locked by any other thread at some point before *abs_time*. Prior *unlock()* operation on the same mutex *synchronizes-with* (as defined in *std::memory_order*) this operation if it returns *true*. If *try_lock_until* is called by a thread that already owns the mutex, the behavior is undefined.

Note: This overload essentially calls

```
bool try_lock_until(
    hpx::chrono::steady_time_point const& abs_time,
    char const* description, error_code& ec = throws);
```

with *description* as *mutex::try_lock_until*.

Parameters

- **abs_time** – time point to block until
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *bool try_lock_until* returns *true* if the lock was acquired successfully, otherwise *false*.

```
inline bool try_lock_for(hpx::chrono::steady_duration const &rel_time, char const *description,
error_code &ec = throws)
```

Tries to lock the mutex. Blocks until specified *rel_time* has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns *true*, otherwise returns *false*. If *rel_time* is less or equal *rel_time.zero()*, the function behaves like *try_lock()*. This function may block for longer than *rel_time* due to scheduling or resource contention delays. As with *try_lock()*, this function is allowed to fail spuriously and return *false* even if the mutex was not locked by any other thread at some point during *rel_time*. Prior *unlock()* operation on the same mutex *synchronizes-with* (as defined in *std::memory_order*) this operation if it returns *true*. If *try_lock_for* is called by a thread that already owns the mutex, the behavior is undefined.

Parameters

- **rel_time** – minimum duration to block for
- **description** – Description of the *timed_mutex*
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *bool try_lock_for* returns *true* if the lock was acquired successfully, otherwise *false*.

```
inline bool try_lock_for(hpx::chrono::steady_duration const &rel_time, error_code &ec = throws)
```

Tries to lock the mutex. Blocks until specified *rel_time* has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns *true*, otherwise returns *false*. If *rel_time* is less or equal *rel_time.zero()*, the function behaves like *try_lock()*. This function may block for longer than *rel_time* due to scheduling or resource contention delays. As with *try_lock()*, this function is allowed to fail spuriously and return *false* even if the mutex was not locked by any other thread at some point during *rel_time*. Prior *unlock()* operation on the same mutex *synchronizes-with* (as defined in *std::memory_order*) this operation if it returns *true*. If *try_lock_for* is called by a thread that already owns the mutex, the behavior is undefined.

Note: This overload essentially calls

```
bool try_lock_for(
    hpx::chrono::steady_duration const& rel_time,
    char const* description, error_code& ec = throws)
```

with *description* as `mutex::try_lock_for`.

Parameters

- **rel_time** – minimum duration to block for
- **ec** – Used to hold error code value originated during the operation. Defaults to `throws` — A special ‘throw on error’ `error_code`.

Returns `bool try_lock_for` returns `true` if the lock was acquired successfully, otherwise `false`.

```
void lock(char const *description, error_code &ec = throws)
```

Locks the mutex. If another thread has already locked the mutex, a call to lock will block execution until the lock is acquired. If lock is called by a thread that already owns the mutex, the behavior is undefined: for example, the program may deadlock. `hpx::mutex` can detect the invalid usage and throws a `std::system_error` with error condition `resource_deadlock_would_occur` instead of deadlock-ing. Prior `unlock()` operations on the same mutex synchronize- with (as defined in `std::memory_order`) this operation.

Note: `lock()` is usually not called directly: `std::unique_lock`, `std::scoped_lock`, and `std::lock_guard` are used to manage exclusive locking.

Parameters

- **description** – Description of the `mutex`
- **ec** – Used to hold error code value originated during the operation. Defaults to `throws` — A special ‘throw on error’ `error_code`.

Returns `void lock` returns `void`.

```
inline void lock(error_code &ec = throws)
```

Locks the mutex. If another thread has already locked the mutex, a call to lock will block execution until the lock is acquired. If lock is called by a thread that already owns the mutex, the behavior is undefined: for example, the program may deadlock. `hpx::mutex` can detect the invalid usage and throws a `std::system_error` with error condition `resource_deadlock_would_occur` instead of deadlock-ing. Prior `unlock()` operations on the same mutex synchronize - with(as defined in `std::memory_order`) this operation.

Note: `lock()` is usually not called directly: `std::unique_lock`, `std::scoped_lock`, and `std::lock_guard` are used to manage exclusive locking. This overload essentially calls `void lock(char const* description, error_code& ec = throws);` with *description* as `mutex::lock`.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to `throws` — A special ‘throw on error’ `error_code`.

Returns `void lock` returns `void`.

```
bool try_lock(char const *description, error_code &ec = throws)
```

Tries to lock the `mutex`. Returns immediately. On successful lock acquisition returns `true`, otherwise returns `false`. This function is allowed to fail spuriously and return `false` even if the `mutex` is not currently locked by any other thread. If `try_lock` is called by a thread that already owns the `mutex`, the behavior is undefined. Prior `unlock()` operation on the same mutex synchronizes-with (as defined in `std::memory_order`) this operation if it returns `true`. Note that prior `lock()` does not synchronize with this operation if it returns `false`.

Parameters

- **description** – Description of the *mutex*
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns bool *try_lock* returns *true* on successful lock acquisition, otherwise returns *false*.

inline bool **try_lock(error_code &ec = throws)**

Tries to lock the *mutex*. Returns immediately. On successful lock acquisition returns *true*, otherwise returns *false*. This function is allowed to fail spuriously and return *false* even if the *mutex* is not currently locked by any other thread. If *try_lock* is called by a thread that already owns the *mutex*, the behavior is undefined. Prior *unlock()* operation on the same *mutex* synchronizes-with (as defined in *std::memory_order*) this operation if it returns *true*. Note that prior *lock()* does not synchronize with this operation if it returns *false*.

Note: This overload essentially calls

```
void try_lock(char const* description,
           error_code& ec = throws);
```

with *description* as *mutex::try_lock*.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns bool *try_lock* returns *true* on successful lock acquisition, otherwise returns *false*.

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

Unlocks the *mutex*. The *mutex* must be locked by the current thread of execution, otherwise, the behavior is undefined. This operation *synchronizes-with* (as defined in *std::memory_order*) any subsequent *lock* operation that obtains ownership of the same *mutex*.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special ‘throw on error’ *error_code*.

Returns *unlock* returns *void*.

namespace **lcos**

namespace **local**

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.

hpx/synchronization/no_mutex.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

struct **no_mutex**

#include <no_mutex.hpp> `no_mutex` class can be used in cases where the shared data between multiple threads can be accessed simultaneously without causing inconsistencies.

Public Functions

inline constexpr void **lock**() noexcept

inline constexpr bool **try_lock**() noexcept

inline constexpr void **unlock**() noexcept

namespace **lcos**

namespace **local**

hpx/synchronization/once.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

⁵¹⁶ https://en.cppreference.com/w/cpp/named_req/Mutex

⁵¹⁷ https://en.cppreference.com/w/cpp/named_req/StandardLayoutType

⁵¹⁸ https://en.cppreference.com/w/cpp/named_req/TimedMutex

⁵¹⁹ https://en.cppreference.com/w/cpp/named_req/StandardLayoutType

Defines

`HPX_ONCE_INIT`

namespace `hpx`

Functions

```
template<typename F, typename ...Args>
void call_once(once_flag &flag, F &&f, Args&&... args)
```

Executes the Callable object *f* exactly once, even if called concurrently, from several threads.

In detail:

- If, by the time *call_once* is called, flag indicates that *f* was already called, *call_once* returns right away (such a call to *call_once* is known as passive).
- Otherwise, *call_once* invokes `std::forward<Callable>(f)` with the arguments `std::forward<Args>(args)...` (as if by `hpx::invoke`). Unlike the `hpx::thread` constructor or `hpx::async`, the arguments are not moved or copied because they don't need to be transferred to another thread of execution. (such a call to *call_once* is known as active).
 - If that invocation throws an exception, it is propagated to the caller of *call_once*, and the flag is not flipped so that another call will be attempted (such a call to *call_once* is known as exceptional).
 - If that invocation returns normally (such a call to *call_once* is known as returning), the flag is flipped, and all other calls to *call_once* with the same flag are guaranteed to be passive. All active calls on the same flag form a single total order consisting of zero or more exceptional calls, followed by one returning call. The end of each active call synchronizes-with the next active call in that order. The return from the returning call synchronizes-with the returns from all passive calls on the same flag: this means that all concurrent calls to *call_once* are guaranteed to observe any side-effects made by the active call, with no additional synchronization.

Note: If concurrent calls to *call_once* pass different functions *f*, it is unspecified which *f* will be called. The selected function runs in the same thread as the *call_once* invocation it was passed to. Initialization of function-local statics is guaranteed to occur only once even when called from multiple threads, and may be more efficient than the equivalent code using `hpx::call_once`. The POSIX equivalent of this function is `pthread_once`.

Parameters

- `flag` – an object, for which exactly one function gets executed
- `f` – Callable object to invoke
- `args...` – arguments to pass to the function

Throws `std::system_error` – if any condition prevents calls to *call_once* from executing as specified or any exception thrown by *f*

struct `once_flag`

`#include <once.hpp>` The class `hpx::once_flag` is a helper structure for `hpx::call_once`. An object of type `hpx::once_flag` that is passed to multiple calls to `hpx::call_once` allows those calls to coordinate

with each other such that only one of the calls will actually run to completion. `hpx::once_flag` is neither copyable nor movable.

Public Functions

`HPX_NON_COPYABLE(once_flag)`

inline `once_flag()` noexcept

Constructs an `once_flag` object. The internal state is set to indicate that no function has been called yet.

Private Members

`std::atomic<long> status_`

`lcos::local::event event_`

Friends

template<typename F, typename ...Args>

friend void `call_once(once_flag &flag, F &&f, Args&&... args)`

Executes the Callable object *f* exactly once, even if called concurrently, from several threads.

In detail:

- If, by the time `call_once` is called, flag indicates that *f* was already called, `call_once` returns right away (such a call to `call_once` is known as passive).
- Otherwise, `call_once` invokes `std::forward<Callable>(f)` with the arguments `std::forward<Args>(args)...` (as if by `hpx::invoke`). Unlike the `hpx::thread` constructor or `hpx::async`, the arguments are not moved or copied because they don't need to be transferred to another thread of execution. (such a call to `call_once` is known as active).
 - If that invocation throws an exception, it is propagated to the caller of `call_once`, and the flag is not flipped so that another call will be attempted (such a call to `call_once` is known as exceptional).
 - If that invocation returns normally (such a call to `call_once` is known as returning), the flag is flipped, and all other calls to `call_once` with the same flag are guaranteed to be passive. All active calls on the same flag form a single total order consisting of zero or more exceptional calls, followed by one returning call. The end of each active call synchronizes-with the next active call in that order. The return from the returning call synchronizes-with the returns from all passive calls on the same flag: this means that all concurrent calls to `call_once` are guaranteed to observe any side-effects made by the active call, with no additional synchronization.

Note: If concurrent calls to `call_once` pass different functions *f*, it is unspecified which *f* will be called. The selected function runs in the same thread as the `call_once` invocation it was passed to. Initialization of function-local statics is guaranteed to occur only once even when called from multiple threads, and may be more efficient than the equivalent code using `hpx::call_once`. The POSIX equivalent of this function is `pthread_once`.

Parameters

- **flag** – an object, for which exactly one function gets executed
- **f** – Callable object to invoke

- **args...** – arguments to pass to the function
- Throws** `std::system_error` – if any condition prevents calls to *call_once* from executing as specified or any exception thrown by *f*

namespace **lcos**

namespace **local**

Functions

```
template<typename F, typename... Args> HPX_DEPRECATED_V (1, 8,
"hpx::lcos::local::call_once is deprecated,
use hpx::call_once " "instead") void call_once(hpx
```

hpx/synchronization/recursive_mutex.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

TypeDefs

```
using recursive_mutex = detail::recursive_mutex_impl<>
```

namespace **lcos**

namespace **local**

hpx/synchronization/shared_mutex.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

TypeDefs

```
using shared_mutex = detail::shared_mutex<>
```

The *shared_mutex* class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In contrast to other mutex types which facilitate exclusive access, a *shared_mutex* has two levels of access:

- *shared* - several threads can share ownership of the same mutex.

- *exclusive* - only one thread can own the mutex.

If one thread has acquired the exclusive lock (through *lock*, *try_lock*), no other threads can acquire the lock (including the shared). If one thread has acquired the shared lock (through *lock_shared*, *try_lock_shared*), no other thread can acquire the exclusive lock, but can acquire the shared lock. Only when the exclusive lock has not been acquired by any thread, the shared lock can be acquired by multiple threads. Within one thread, only one lock (shared or exclusive) can be acquired at the same time. Shared mutexes are especially useful when shared data can be safely read by any number of threads simultaneously, but a thread may only write the same data when no other thread is reading or writing at the same time. The *shared_mutex* class satisfies all requirements of *SharedMutex* and *StandardLayoutType*.

namespace **lcos**

namespace **local**

hpx/synchronization/sliding_semaphore.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Typedefs

```
using sliding_semaphore = sliding_semaphore_var<>
template<typename Mutex = hpx::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

inline **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.

inline 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.

inline 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.

inline bool **try_wait**(*std*::int64_t upper_limit = 1)

Try to 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.

Returns The function returns true if the calling thread would not block if it was calling *wait()*.

inline 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.

inline *std*::int64_t **signal_all**()

Private Types

using **mutex_type** = *Mutex*

Private Members

mutable *mutex_type* **mtx_**

lcos::local::detail::sliding_semaphore **sem_**

namespace **lcos**

namespace **local**

hpx/synchronization/spinlock.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Typedefs

using **spinlock** = detail::spinlock<true>

spinlock is a type of lock that causes a thread attempting to obtain it to check for its availability while waiting in a loop continuously.

using **spinlock_no_backoff** = detail::spinlock<false>

namespace **lcos**

namespace **local**

hpx/synchronization/stop_token.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

template<typename **Callback**>

stop_callback(*stop_token*, *Callback*) -> *stop_callback*<*Callback*>

The **stop_callback** class template provides an RAII object type that registers a callback function for an associated [hpx::stop_token](#) object, such that the callback function will be invoked when the [hpx::stop_token](#)'s associated [hpx::stop_source](#) is requested to stop. Callback functions registered via **stop_callback**'s constructor are invoked either in the same thread that successfully invokes *request_stop()* for a [hpx::stop_source](#) of the **stop_callback**'s associated [hpx::stop_token](#); or if stop has already been requested prior to the constructor's registration, then the callback is invoked in the thread constructing the **stop_callback**. More than one **stop_callback** can be created for the same [hpx::stop_token](#), from the same or different threads concurrently. No guarantee is provided for the order in which they will be executed, but they will be invoked synchronously; except for *stop_callback(s)* constructed after stop has already been requested for the [hpx::stop_token](#), as described previously. If an invocation of a callback exits via an exception then [hpx::terminate](#) is called. [hpx::stop_callback](#) is not *CopyConstructible*, *CopyAssignable*, *MoveConstructible*, nor *MoveAssignable*. The template param *Callback* type must be both *invocable* and *destructible*. Any return value is ignored.

inline void **swap**(*stop_token* &lhs, *stop_token* &rhs) noexcept

inline void **swap**(*stop_source* &lhs, *stop_source* &rhs) noexcept

Variables

```
constexpr nostopstate_t nostopstate = {}
```

This is a constant object instance of `hpx::nostopstate_t` for use in constructing an empty `hpx::stop_source`, as a placeholder value in the non-default constructor.

```
struct nostopstate_t
```

`#include <stop_token.hpp>` Unit type intended for use as a placeholder in `hpx::stop_source` non-default constructor, that makes the constructed `hpx::stop_source` empty with no associated stop-state.

Public Functions

```
explicit nostopstate_t() = default
```

```
template<typename Callback>
```

```
class stop_callback
```

```
class stop_source
```

`#include <stop_token.hpp>` The `stop_source` class provides the means to issue a stop request, such as for `hpx::jthread` cancellation. A stop request made for one `stop_source` object is visible to all `stop_sources` and `hpx::stop_tokens` of the same associated stop-state; any `hpx::stop_callback(s)` registered for associated `hpx::stop_token(s)` will be invoked, and any `hpx::condition_variable_any` objects waiting on associated `hpx::stop_token(s)` will be awoken. Once a stop is requested, it cannot be withdrawn. Additional stop requests have no effect.

Note: For the purposes of `hpx::jthread` cancellation the `stop_source` object should be retrieved from the `hpx::jthread` object using `get_stop_source()`; or stop should be requested directly from the `hpx::jthread` object using `request_stop()`. This will then use the same associated stop-state as that passed into the `hpx::jthread`'s invoked function argument (i.e., the function being executed on its thread). For other uses, however, a `stop_source` can be constructed separately using the default constructor, which creates new stop-state.

Public Functions

```
inline stop_source()
```

```
inline explicit stop_source(nostopstate_t) noexcept
```

```
inline stop_source(stop_source const &rhs) noexcept
```

```
stop_source(stop_source&&) noexcept = default
```

```
inline stop_source &operator=(stop_source const &rhs) noexcept
```

```
stop_source &operator=(stop_source&&) noexcept = default
```

```
inline ~stop_source()
```

```
inline void swap(stop_source &s) noexcept
    swaps two stop_source objects

inline stop_token get_token() const noexcept
    returns a stop_token for the associated stop-state

inline bool stop_possible() const noexcept
    checks whether associated stop-state can be requested to stop

inline bool stop_requested() const noexcept
    checks whether the associated stop-state has been requested to stop

inline bool request_stop() noexcept
    makes a stop request for the associated stop-state, if any
```

Private Members

hpx::intrusive_ptr<detail::stop_state> **state_**

Friends

```
inline friend bool operator==(stop_source const &lhs, stop_source const &rhs) noexcept
inline friend bool operator!=(stop_source const &lhs, stop_source const &rhs) noexcept
```

class **stop_token**

#include <stop_token.hpp> The **stop_token** class provides the means to check if a stop request has been made or can be made, for its associated **hpx::stop_source** object. It is essentially a thread-safe “view” of the associated stop-state. The **stop_token** can also be passed to the constructor of **hpx::stop_callback**, such that the callback will be invoked if the **stop_token**’s associated **hpx::stop_source** is requested to stop. And **stop_token** can be passed to the interruptible waiting functions of **hpx::condition_variable_any**, to interrupt the condition variable’s wait if stop is requested.

Note: A **stop_token** object is not generally constructed independently, but rather retrieved from a **hpx::jthread** or **hpx::stop_source**. This makes it share the same associated stop-state as the **hpx::jthread** or **hpx::stop_source**.

Public Types

```
template<typename Callback>
using callback_type = stop_callback<Callback>
```

Public Functions

```
stop_token() noexcept = default
inline stop_token(stop_token const &rhs) noexcept
stop_token(stop_token&&) noexcept = default
inline stop_token &operator=(stop_token const &rhs) noexcept
stop_token &operator=(stop_token&&) noexcept = default
~stop_token() = default
inline void swap(stop_token &s) noexcept
    swaps two stop_token objects
inline bool stop_requested() const noexcept
    checks whether the associated stop-state has been requested to stop
inline bool stop_possible() const noexcept
    checks whether associated stop-state can be requested to stop
```

Private Functions

```
inline explicit stop_token(hpx::intrusive_ptr<detail::stop_state> const &state) noexcept
```

Private Members

```
hpx::intrusive_ptr<detail::stop_state> state_
```

Friends

```
friend class stop_callback
friend class stop_source
inline friend bool operator==(stop_token const &lhs, stop_token const &rhs) noexcept
inline friend bool operator!=(stop_token const &lhs, stop_token const &rhs) noexcept
namespace experimental
namespace p2300_stop_token
```

Functions

```
template<typename Callback>
in_place_stop_callback(in_place_stop_token, Callback) -> in_place_stop_callback<Callback>

template<typename Callback>
class in_place_stop_callback

class in_place_stop_source
```

Public Functions

```
inline in_place_stop_source() noexcept
inline ~in_place_stop_source()
in_place_stop_source(in_place_stop_source const&) = delete
in_place_stop_source(in_place_stop_source&&) noexcept = delete
in_place_stop_source &operator=(in_place_stop_source const&) = delete
in_place_stop_source &operator=(in_place_stop_source&&) noexcept = delete
inline in_place_stop_token get_token() const noexcept
inline bool request_stop() noexcept
inline bool stop_requested() const noexcept
inline bool stop_possible() const noexcept
```

Private Functions

```
inline bool register_callback(hpx::detail::stop_callback_base *cb) noexcept
inline void remove_callback(hpx::detail::stop_callback_base *cb) noexcept
```

Private Members

hpx::detail::stop_state **state_**

Friends

```
friend class in_place_stop_token
friend class in_place_stop_callback

class in_place_stop_token
```

Public Types

```
template<typename Callback>
using callback_type = in_place_stop_callback<Callback>
```

Public Functions

```
inline constexpr in_place_stop_token() noexcept
in_place_stop_token(in_place_stop_token const &rhs) noexcept = default
inline in_place_stop_token(in_place_stop_token &&rhs) noexcept
in_place_stop_token &operator=(in_place_stop_token const &rhs) noexcept = default
inline in_place_stop_token &operator=(in_place_stop_token &&rhs) noexcept
inline bool stop_requested() const noexcept
inline bool stop_possible() const noexcept
inline void swap(in_place_stop_token &rhs) noexcept
```

Private Functions

```
inline explicit in_place_stop_token(in_place_stop_source const *source) noexcept
```

Private Members

```
in_place_stop_source const *source_
```

Friends

```
friend class in_place_stop_source
friend class in_place_stop_callback

inline friend constexpr friend bool operator== (in_place_stop_token const &lhs,
in_place_stop_token const &rhs) noexcept

inline friend void swap(in_place_stop_token &x, in_place_stop_token &y) noexcept

struct never_stop_token
```

Public Types

```
template<typename>
using callback_type = callback_impl
```

Public Static Functions

```
static inline constexpr bool stop_requested() noexcept
static inline constexpr bool stop_possible() noexcept
```

Friends

```
inline friend constexpr friend bool operator== (never_stop_token const &,  
never_stop_token const &) noexcept
```

```
struct callback_impl
```

Public Functions

```
template<typename Callback>
inline explicit constexpr callback_impl (never_stop_token, Callback&&) noexcept
```

tag_invoke

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/functional/traits/is_invocable.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Variables

```
template<typename F, typename ...Tsis_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

template<typename F, typename ...Ts>
constexpr bool is_nothrow_invocable_v = is_nothrow_invocable<F, Ts...>::value

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

```
struct is_invocable : public hpx::detail::is_invocable_impl<F&&(Ts&&...), void>
```

#include <is_invocable.hpp> Determines whether *F* can be invoked with the arguments *Ts*.... Formally, determines whether.

```
(INVOKE(std::declval<F>(), std::declval<Ts>()...))
```

is well formed when treated as an unevaluated operand, where *INVOKE* is the operation defined in *Callable*.

F, *R* and all types in the parameter pack *Ts* shall each be a complete type, (possibly cv-qualified) void, or an array of unknown bound. Otherwise, the behavior is undefined. If an instantiation of a template above depends, directly or indirectly, on an incomplete type, and that instantiation could yield a different result if that type were hypothetically completed, the behavior is undefined.

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

```
struct is_invocable_r : public hpx::detail::is_invocable_r_impl<F&&(Ts&&...), R>
```

#include <is_invocable.hpp> Determines whether *F* can be invoked with the arguments *Ts*... to yield a result that is convertible to *R* and the implicit conversion does not bind a reference to a temporary object (since C++23). If *R* is cv void, the result can be any type. Formally, determines whether

```
(INVOKE<R>(std::declval<F>(), std::declval<Ts>()...))
```

is well formed when treated as an unevaluated operand, where *INVOKE* is the operation defined in *Callable*.

```
(INVOKE(std::declval<F>(), std::declval<Ts>()...))
```

is well formed when treated as an unevaluated operand, where *INVOKE* is the operation defined in *Callable*.

F, *R* and all types in the parameter pack *Ts* shall each be a complete type, (possibly cv-qualified) void, or an array of unknown bound. Otherwise, the behavior is undefined. If an instantiation of a template above depends, directly or indirectly, on an incomplete type, and that instantiation could yield a different result if that type were hypothetically completed, the behavior is undefined.

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

```
struct is_nothrow_invocable : public hpx::detail::is_nothrow_invocable_impl<F(Ts...), is_invocable_v<F, Ts...>>
```

thread_pool_util

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/thread_pool_util/thread_pool_suspension_helpers.hpp

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.

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

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

`void resume_processing_unit_cb(thread_pool_base &pool, hpx::function<void(void)> callback, std::size_t virt_core, error_code &ec = throws)`

Resumes the given processing unit. Takes a callback as a parameter which will be called when the processing unit has been resumed.

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

Parameters

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

`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.

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

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

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

`void suspend_processing_unit_cb(hpx::function<void(void)> callback, thread_pool_base &pool, std::size_t virt_core, error_code &ec = throws)`

Suspends the given processing unit. Takes a callback as a parameter which will be called when the processing unit has been suspended.

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

Parameters

- `callback` – [in] Callback which is called when the processing unit has been suspended.
- `virt_core` – [in] The processing unit to suspend.

- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

`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.

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

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

`void resume_pool_cb(thread_pool_base &pool, hpx::function<void(void)> callback, error_code &ec = throws)`

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

Parameters

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

`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.

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

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

`void suspend_pool_cb(thread_pool_base &pool, hpx::function<void(void)> callback, error_code &ec = throws)`

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

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

Parameters

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

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

threading

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/threading/jthread.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

inline void **swap**(*jthread* &lhs, *jthread* &rhs) noexcept

class **jthread**

#include <jthread.hpp> The class *jthread* represents a single thread of execution. It has the same general behavior as *hpx::thread*, except that *jthread* automatically rejoins on destruction, and can be cancelled/stopped in certain situations. Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, *hpx::terminate* is called. The top-level function may communicate its return value or an exception to the caller via *hpx::promise* or by modifying shared variables (which may require synchronization, see *hpx::mutex* and *hpx::atomic*) Unlike *hpx::thread*, the *jthread* logically holds an internal private member of type *hpx::stop_source*, which maintains a shared stop-state. The *jthread* constructor accepts a function that takes a *hpx::stop_token* as its first argument, which will be passed in by the *jthread* from its internal *stop_source*. This allows the function to check if stop has been requested during its execution, and return if it has. *hpx::jthread* objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may be not associated with any *jthread* objects (after detach). No two *hpx::jthread* objects may represent the same thread of execution; *hpx::jthread* is not *CopyConstructible* or *CopyAssignable*, although it is *MoveConstructible* and *MoveAssignable*.

Public Types

using **id** = *thread::id*

using **native_handle_type** = *thread::native_handle_type*

Public Functions

inline **jthread**() noexcept

template<typename F, typename ...Ts, typename **Enable** = typename
*std::enable_if<!std::is_same<typename std::decay<F>::type, jthread>::value>::type>
inline explicit **jthread**(F &&f, Ts&&... ts)*

inline **~jthread**()

jthread(*jthread* const&) = delete

```

jthread(jthread &&x) noexcept = default
jthread &operator=(jthread const&) = delete
jthread &operator=(jthread&&) noexcept = default
    moves the jthread object
inline void swap(jthread &t) noexcept
    swaps two jthread objects
inline bool joinable() const noexcept
    checks whether the thread is joinable, i.e. potentially running in parallel context
inline void join()
    waits for the thread to finish its execution
inline void detach()
    permits the thread to execute independently from the thread handle
inline id get_id() const noexcept
    returns the id of the thread
inline native_handle_type native_handle()
    returns the underlying implementation-defined thread handle
inline stop_source get_stop_source() noexcept
    returns a stop_source object associated with the shared stop state of the thread
inline stop_token get_stop_token() const noexcept
    returns a stop_token associated with the shared stop state of the thread
inline bool request_stop() noexcept
    requests execution stop via the shared stop state of the thread

```

Public Static Functions

```

static inline unsigned int hardware_concurrency()
    returns the number of concurrent threads supported by the implementation

```

Private Members

stop_source sssource_

hpx::thread thread_ = {}

Private Static Functions

```
template<typename F, typename ...Ts>
static inline void invoke(std::false_type, F &&f, stop_token&&, Ts&&... ts)

template<typename F, typename ...Ts>
static inline void invoke(std::true_type, F &&f, stop_token &&st, Ts&&... ts)
```

hpx/threading/thread.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

```
template<>

struct std::hash<::hpx::thread::id>
```

Public Functions

```
inline std::size_t operator() (::hpx::thread::id const &id) const
```

```
namespace hpx
```

TypeDefs

```
using thread_termination_handler_type = hpx::function<void(std::exception_ptr const &e)>
```

Functions

```
void set_thread_termination_handler(thread_termination_handler_type f)

inline void swap(thread &x, thread &y) noexcept

inline bool operator==(thread::id const &x, thread::id const &y) noexcept

inline bool operator!=(thread::id const &x, thread::id const &y) noexcept

inline bool operator<(thread::id const &x, thread::id const &y) noexcept

inline bool operator>(thread::id const &x, thread::id const &y) noexcept

inline bool operator<=(thread::id const &x, thread::id const &y) noexcept

inline bool operator>=(thread::id const &x, thread::id const &y) noexcept

template<typename Char, typename Traits>
std::basic_ostream<Char, Traits> &operator<<(std::basic_ostream<Char, Traits> &out, thread::id const &id)
```

class **thread**

`#include <thread.hpp>` The class `thread` represents a single thread of execution. Threads allow multiple functions to execute concurrently. threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, `hpx::terminate` is called. The top-level function may communicate its return value or an exception to the caller via `hpx::promise` or by modifying shared variables (which may require synchronization, see `hpx::mutex` and `hpx::atomic`) `hpx::thread` objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may not be associated with any thread objects (after detach). No two `hpx::thread` objects may represent the same thread of execution; `hpx::thread` is not *CopyConstructible* or *CopyAssignable*, although it is *MoveConstructible* and *MoveAssignable*.

Public Types

`typedef threads::thread_id_type native_handle_type`

Public Functions

thread() noexcept

template<typename F, typename **Enable** = typename `std`::enable_if<!`std`::is_same<typename `std`::decay<`F`>::type, `thread`>::value>::type>
inline explicit **thread**(`F` &&f)

template<typename F, typename ...Ts>
inline explicit **thread**(`F` &&f, `Ts`&&... vs)

template<typename F>
inline **thread**(`threads::thread_pool_base` *pool, `F` &&f)

template<typename F, typename ...Ts>
inline **thread**(`threads::thread_pool_base` *pool, `F` &&f, `Ts`&&... vs)

~thread()

thread(`thread`&&) noexcept

`thread` &**operator=(**`thread`&&)`)` noexcept

void **swap**(`thread`&) noexcept

swaps two thread objects

inline bool **joinable**() const noexcept

checks whether the thread is joinable, i.e. potentially running in parallel context

void **join()**

waits for the thread to finish its execution

inline void **detach()**

permits the thread to execute independently from the thread handle

```
id get_id() const noexcept
    returns the id of the thread

inline native_handle_type native_handle() const
    returns the underlying implementation-defined thread handle

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 unsigned int hardware_concurrency() noexcept
    returns the number of concurrent threads supported by the implementation

static void interrupt(id, bool flag = true)
```

Private Types

```
typedef hpx::spinlock mutex_type
```

Private Functions

```
void terminate(const char *function, const char *reason) const

inline bool joinable_locked() const noexcept

inline void detach_locked()

void start_thread(threads::thread_pool_base *pool, hpx::move_only_function<void()> &&func)
```

Private Members

```
mutable mutex_type mtx_
```

```
threads::thread_id_ref_type id_
```

Private Static Functions

```
static threads::thread_result_type thread_function_nullary(hpx::move_only_function<void()> const &func)
```

class **id**

Public Functions

```
id() noexcept = default

inline explicit id(threads::thread_id_type const &i) noexcept
inline explicit id(threads::thread_id_type &&i) noexcept
inline explicit id(threads::thread_id_ref_type const &i) noexcept
inline explicit id(threads::thread_id_ref_type &&i) noexcept
inline threads::thread_id_type const &native_handle() const
```

Private Members

threads::*thread_id_type* **id_**

Friends

```
friend class thread

friend bool operator==(thread::id const &x, thread::id const &y) noexcept
friend bool operator!=(thread::id const &x, thread::id const &y) noexcept
friend bool operator<(thread::id const &x, thread::id const &y) noexcept
friend bool operator>(thread::id const &x, thread::id const &y) noexcept
friend bool operator<=(thread::id const &x, thread::id const &y) noexcept
friend bool operator>=(thread::id const &x, thread::id const &y) noexcept

template<typename Char, typename Traits>
friend std::basic_ostream<Char, Traits> &operator<<(std::basic_ostream<Char, Traits>&,
```

namespace **this** **thread**

Functions

`thread::id get_id()` noexcept

Returns the id of the current thread.

`void yield()` noexcept

Provides a hint to the implementation to reschedule the execution of threads, allowing other threads to run.

Note: The exact behavior of this function depends on the implementation, in particular on the mechanics of the OS scheduler in use and the state of the system. For example, a first-in-first-out realtime scheduler (SCHED_FIFO in Linux) would suspend the current thread and put it on the back of the queue of the same-priority threads that are ready to run (and if there are no other threads at the same priority, `yield` has no effect).

`void yield_to(thread::id)` noexcept

`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)`

Blocks the execution of the current thread until specified `abs_time` has been reached.

It is recommended to use the clock tied to `abs_time`, in which case adjustments of the clock may be taken into account. Thus, the duration of the block might be more or less than `abs_time-Clock::now()` at the time of the call, depending on the direction of the adjustment and whether it is honored by the implementation. The function also may block until after `abs_time` has been reached due to process scheduling or resource contention delays.

Parameters `abs_time` – absolute time to block until

`inline void sleep_for(hpx::chrono::steady_duration const &rel_time)`

Blocks the execution of the current thread for at least the specified `rel_time`. This function may block for longer than `rel_time` due to scheduling or resource contention delays.

It is recommended to use a steady clock to measure the duration. If an implementation uses a system clock instead, the wait time may also be sensitive to clock adjustments.

Parameters `rel_time` – time duration to sleep

`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 class restore_interruption  
class restore_interruption
```

Public Functions

```
explicit 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<> id >
```

Public Functions

```
inline 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.

hpx/threading_base/annotated_function.hpp

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) noexcept
```

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&) noexcept
```

namespace **traits**

namespace **util**

Functions

```
template<typename F>
constexpr decltype(auto) annotated_function(F &&f, char const *name = nullptr) noexcept
```

```
template<typename F>
constexpr decltype(auto) annotated_function(F &&f, std::string const &name) noexcept
```

hpx/threading_base/print.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/threading_base/register_thread.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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)

inline 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.

Note: As long as *ec* is not pre-initialized to *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.

Throws *invalid_status* – if the runtime system has not been started yet.

Returns This function will return the internal id of the newly created HPX-thread.

```
inline 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.

Note: As long as *ec* is not pre-initialized to *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.

Throws *invalid_status* – if the runtime system has not been started yet.

Returns This function will return the internal id of the newly created HPX-thread.

```
inline 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.

Throws `invalid_status` – if the runtime system has not been started yet.

inline `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.

Throws `invalid_status` – if the runtime system has not been started yet.

hpx/threading_base/scoped_annotation.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

struct **scoped_annotation**

#include <`scoped_annotation.hpp`> `scoped_annotation` associates a `name` with a section of code (scope). It can be used to visualize code execution in profiling tools like *Intel VTune*, *Apex Profiler*, etc. That allows analysing performance to figure out which part(s) of code is (are) responsible for performance degradation, etc.

Public Functions

HPX_NON_COPYABLE(scoped_annotation)

inline explicit constexpr `scoped_annotation`(char const*) noexcept

template<typename **F**>

inline explicit constexpr `scoped_annotation`(**F**&&) noexcept

inline `~scoped_annotation()`

namespace **util**

Typedefs

using **instead** = *hpx::scoped_annotation*

hpx/threading_base/thread_data.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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`.

Public Types

```
using spinlock_pool = util::spinlock_pool<thread_data>
```

Public Functions

```
thread_data(thread_data const&) = delete
thread_data(thread_data&&) = delete
thread_data &operator=(thread_data const&) = delete
thread_data &operator=(thread_data&&) = delete
inline thread_state get_state(std::memory_order order = std::memory_order_acquire) const
    noexcept
```

The get_state function queries the state of this thread instance.

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*.

Returns This function returns the current state of this thread. It will return one of the values as defined by the *thread_state* enumeration.

```
inline 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) noexcept
```

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.

```
inline 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) noexcept
```

```
inline 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) noexcept
```

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.

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.

Returns This function returns *true* if the state has been changed successfully

```
inline 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) noexcept
```

```
inline constexpr std::uint64_t get_component_id() const noexcept
```

Return the id of the component this thread is running in.

```
inline util::thread_description get_description() const
```

```
inline util::thread_description set_description(util::thread_description)
```

```
inline util::thread_description get_lco_description() const
```

```
inline util::thread_description set_lco_description(util::thread_description)
```

```
inline constexpr std::uint32_t get_parent_locality_id() const noexcept
```

Return the locality of the parent thread.

```
inline constexpr thread_id_type get_parent_thread_id() const noexcept
```

Return the thread id of the parent thread.

```
inline constexpr std::size_t get_parent_thread_phase() const noexcept
```

Return the phase of the parent thread.

```
inline constexpr util::backtrace const *get_backtrace() const noexcept
```

```
inline util::backtrace const *set_backtrace(util::backtrace const*) noexcept
```

```
inline constexpr thread_priority get_priority() const noexcept
```

```
inline void set_priority(thread_priority priority) noexcept
```

```
inline bool interruption_requested() const noexcept
```

```
inline bool interruption_enabled() const noexcept
```

```
inline bool set_interruption_enabled(bool enable) noexcept
```

```
inline void interrupt(bool flag = true)
```

```
bool interruption_point(bool throw_on_interrupt = true)
```

```
bool add_thread_exit_callback(function<void()> const &f)
```

```
void run_thread_exit_callbacks()
```

```
void free_thread_exit_callbacks()
```

```
inline constexpr bool is_stackless() const noexcept
```

```
void destroy_thread() override
```

```

inline constexpr policies::scheduler_base *get_scheduler_base() const noexcept
inline constexpr std::size_t get_last_worker_thread_num() const noexcept
inline void set_last_worker_thread_num(std::size_t last_worker_thread_num) noexcept
inline constexpr std::ptrdiff_t get_stack_size() const noexcept
inline thread_stacksize get_stack_size_enum() const noexcept
template<typename ThreadQueue>
inline constexpr ThreadQueue &get_queue() noexcept
inline coroutine_type::result_type operator() (hpx::execution_base::this_thread::detail::agent_storage
*agent_storage)

Execute the thread function.
Returns 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.

inline virtual thread_id_type get_thread_id() const
inline virtual std::size_t get_thread_phase() const noexcept
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() override
virtual void destroy() noexcept = 0

```

Protected Functions

```
inline thread_restart_state set_state_ex(thread_restart_state new_state) noexcept
```

The set_state function changes the extended state of this thread instance.

Note: This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager.

Parameters **newstate** – [in] The new extended state to be set for the thread.

```
void rebind_base(thread_init_data &init_data)
```

Private Members

```
mutable 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<hpx::function<void()>> exit_funcs_

scheduler_base_

std::size_t last_worker_thread_num_

std::ptrdiff_t stacksize_

thread_stacksize stacksize_enum_

void *queue_
```

[hpx/threading_base/thread_description.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

Note: As long as `ec` is not pre-initialized to `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.

Returns 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>”.

```
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:

```
enumerator data_type_description
```

```
enumerator data_type_address
```

Public Functions

```
thread_description() noexcept = default
```

```
inline constexpr thread_description(char const*) noexcept
```

```
inline explicit constexpr thread_description(std::string const&) noexcept
```

```
template<typename F, typename = typename std::enable_if<!std::is_same<F,
thread_description>::value && !traits::is_action<F>::value>::type>
```

```
inline explicit constexpr thread_description(F const&, char const* = nullptr) noexcept
template<typename Action, typename = typename
         std::enable_if<traits::is_action<Action>::value>::type>
inline explicit constexpr thread_description(Action, char const* = nullptr) noexcept
inline constexpr data_type kind() const noexcept
inline constexpr char const *get_description() const noexcept
inline constexpr std::size_t get_address() const noexcept
inline explicit constexpr operator bool() const noexcept
inline constexpr bool valid() const noexcept
```

Private Functions

```
void init_from_alternative_name(char const *altname)
```

hpx/threading_base/thread_helpers.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **this_thread**

Functions

```
threads::thread_restart_state suspend(threads::thread_schedule_state state, threads::thread_id_type id,
                                         util::thread_description const &description =
                                         util::thread_description("this_thread::suspend"), error_code &ec
                                         = throws)
```

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to the thread state passed as the parameter.

Note: Must be called from within a HPX-thread.

Throws 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*.

```
inline 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.

Throws 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.

Throws 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*.

```
inline 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.

Throws 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*.

```
inline 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.

Throws 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`.

```
inline 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"),
                                             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.

Throws 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`.

```
inline threads::thread_restart_state suspend(std::uint64_t ms, util::thread_description const
                                             &description =
                                             util::thread_description("this_thread::suspend"),
                                             error_code &ec = throws)
```

The function `suspend` will return control to the thread manager (suspends the current thread). It sets the new state of this thread to `suspended` and schedules a wakeup for this threads after the given time (specified in milliseconds).

Note: Must be called from within a HPX-thread.

Throws 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

Throws 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, hpx::error_code
                           &ec = throws)
```

Set the thread state of the *thread* referenced by the thread_id *id*.

Note: If the thread referenced by the parameter *id* is in *thread_state::active* state this function schedules a new thread which will set the state of the thread as soon as its not active anymore. The function returns *thread_state::active* in this case.

Note: As long as *ec* is not pre-initialized to *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.

Returns 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*.

```
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)

Note: As long as *ec* is not pre-initialized to *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.

Returns

```
inline 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)  
  
inline 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)

Note: As long as *ec* is not pre-initialized to *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.

Returns

```
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).

Note: As long as *ec* is not pre-initialized to *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.

Note: As long as *ec* is not pre-initialized to *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.
- **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.

Returns 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.

Returns 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*.

`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.

Note: As long as *ec* is not pre-initialized to `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.

Returns 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`.

`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.

Note: As long as *ec* is not pre-initialized to `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.

Returns This function returns `true` if the given thread can be interrupted at this point in time. It will return `false` otherwise.

`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.

Note: As long as *ec* is not pre-initialized to `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.

Returns This function returns the previous value of whether the given thread could have been interrupted.

bool **get_thread_interruption_requested**(thread_id_type const &id, *error_code* &ec = `throws`)

Returns whether the given thread has been flagged 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 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.

Returns This function returns *true* if the given thread was flagged for interruption. It will return *false* otherwise.

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.

inline 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

Throws 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`.

hpx/threading_base/thread_num_tss.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 **threads**

hpx/threading_base/thread_pool_base.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

namespace **threads**

Functions

`std::ostream &operator<<(std::ostream &os, thread_pool_base const &thread_pool)`

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.

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

struct `thread_pool_init_parameters`

Public Functions

```
inline 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_
```

hpx/threading_base/threading_base_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **threads**

 namespace **policies**

threadmanager

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/modules/threadmanager.hpp

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 &rtcfg_, notification_policy_type &notifier,  
    detail::network_background_callback_type network_background_callback =  
    detail::network_background_callback_type())
```

```
~threadmanager()
```

```
void init()
```

```
void create_pools()
```

```
void print_pools(std::ostream&)
```

FIXME move to private and add `hpx:printpools` cmd line option.

```
thread_pool_base &default_pool() const
```

```
scheduler_type &default_scheduler() const
```

```
thread_pool_base &get_pool(std::string const &pool_name) const
```

```
thread_pool_base &get_pool(pool_id_type const &pool_id) const
```

```
thread_pool_base &get_pool(std::size_t thread_index) const
```

```
bool pool_exists(std::string const &pool_name) const
```

```
bool pool_exists(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*.

Returns 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()
```

```
inline 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(hpx::function<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)

inline 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.

inline std::thread &get_os_thread_handle(std::size_t num_thread) const

inline 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.).

inline mask_type get_used_processing_units() const
    Returns the mask identifying all processing units used by this thread manager.

inline hwloc_bitmap_ptr get_pool_numa_bitmap(const std::string &pool_name) const

inline void set_scheduler_mode(threads::policies::scheduler_mode mode)
inline void add_scheduler_mode(threads::policies::scheduler_mode mode)
inline void add_remove_scheduler_mode(threads::policies::scheduler_mode to_add_mode,
                                         threads::policies::scheduler_mode to_remove_mode)

inline void remove_scheduler_mode(threads::policies::scheduler_mode mode)
inline void reset_thread_distribution()
inline void init_tss(std::size_t global_thread_num)
inline void deinit_tss()

std::int64_t get_queue_length(bool reset)
std::int64_t get_cumulative_duration(bool reset)

inline std::int64_t get_thread_count_unknown(bool reset)
inline std::int64_t get_thread_count_active(bool reset)
inline std::int64_t get_thread_count_pending(bool reset)
inline std::int64_t get_thread_count_suspended(bool reset)
inline std::int64_t get_thread_count_terminated(bool reset)
inline std::int64_t get_thread_count_staged(bool reset)
```

Private Types

typedef *std*::mutex **mutex_type**

Private Members

mutable *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*.

hpx/timed_execution/timed_execution.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

 namespace **parallel**

 namespace **execution**

hpx/timed_execution/timed_execution_fwd.hpp

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::tagFallback<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.

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.

Param exec [in] The executor object to use for scheduling of the function *f*.

Param rel_time [in] The duration of time after which the given function should be scheduled to run.

Param f [in] The function which will be scheduled using the given executor.

Param ts... [in] Additional arguments to use to invoke *f*.

Return *f(ts...)*'s result through a future

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
inline decltype(auto) friend tagFallback_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::tagFallback<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.

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.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param abs_time [in] The point in time the given function should be scheduled at to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts... [in] Additional arguments to use to invoke *f*.
Return *f*(ts...)’s result through a future

Private Functions

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

Param exec [in] The executor object to use for scheduling of the function *f*.
Param rel_time [in] The duration of time after which the given function should be scheduled to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts... [in] Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
inline decltype(auto) friend tag_fallback_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::tag_fallback<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.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param abs_time [in] The point in time the given function should be scheduled at to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts... [in] Additional arguments to use to invoke *f*.

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(post_at_t, Executor &&exec,
                                               hpx::chrono::steady_time_point const
                                               &abs_time, F &&f, Ts&&... ts)

struct sync_execute_after_t : public
hpx::functional::detail::tag_fallback<sync_execute_after_t>
#include <timed_execution_fwd.hpp> Customization point of synchronous execution agent cre-
ation supporting timed execution.
```

This synchronously creates a single function invocation *f()* using the associated executor at the given point in time.

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.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param rel_time [in] The duration of time after which the given function should be sched-
uled to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts... [in] Additional arguments to use to invoke *f*.
Return *f(ts...)*'s result

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
inline 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 cre-
ation supporting timed execution.
```

This synchronously creates a single function invocation *f()* using the associated executor at the given point in time.

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.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param abs_time [in] The point in time the given function should be scheduled at to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts... [in] Additional arguments to use to invoke *f*.
Return *f(ts...)*'s result

Private Functions

```
template<typename Executor, typename F, typename ...Ts>
inline decltype(auto) friend tag_fallback_invoke(sync_execute_at_t, Executor &&exec,
                                                 hpx::chrono::steady_time_point const
                                                 &abs_time, F &&f, Ts&&... ts)

template<typename BaseExecutor>
struct timed_executor
```

hpx/timed_execution/timed_executors.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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
```

hpx/timed_execution/traits/is_timed_executor.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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
template<typename T>
struct is_timed_executor : public detail::is_timed_executor<std::decay_t<T>>
```

namespace **traits**

```
template<typename Executor, typename Enable = void>
struct is_timed_executor : public hpx::parallel::execution::is_timed_executor<Executor>
```

timing

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/timing/high_resolution_clock.hpp

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**

```
#include <high_resolution_clock.hpp> Class hpx::chrono::high_resolution_clock represents
the clock with the smallest tick period provided by the implementation. It may be an alias of
std::chrono::system_clock or std::chrono::steady_clock, or a third, independent clock.
hpx::chrono::high_resolution_clock meets the requirements of TrivialClock.
```

Public Static Functions

```
static inline std::uint64_t now() noexcept
    returns a std::chrono::time_point representing the current value of the clock

static inline constexpr std::uint64_t() min () noexcept

static inline constexpr std::uint64_t() max () noexcept
```

hpx/timing/high_resolution_timer.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **chrono**

class **high_resolution_timer**

#include <high_resolution_timer.hpp> **high_resolution_timer** is a timer object which measures the elapsed time

Public Types

enum class **init**

Values:

enumerator **no_init**

Public Functions

inline **high_resolution_timer()** noexcept

inline explicit **constexpr high_resolution_timer(*init*) noexcept**

inline explicit **constexpr high_resolution_timer(double t) noexcept**

inline void **restart()** noexcept

restarts the timer

inline double **elapsed()** const noexcept

returns the elapsed time in seconds

inline *std*::int64_t **elapsed_microseconds()** const noexcept

returns the elapsed time in microseconds

inline *std*::int64_t **elapsed_nanoseconds()** const noexcept

returns the elapsed time in nanoseconds

Public Static Functions

```
static inline double now() noexcept
    returns the current time

static inline constexpr double elapsed_max() noexcept
    returns the estimated maximum value for elapsed()

static inline constexpr double elapsed_min() noexcept
    returns the estimated minimum value for elapsed()
```

Protected Static Functions

```
static inline std::uint64_t take_time_stamp() noexcept
```

Private Members

```
std::uint64_t start_time_
```

topology

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/topology/cpu_mask.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

```
namespace threads
```

hpx/topology/topology.hpp

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:

enumerator **membind_default**

enumerator **membind_firstrtouch**

enumerator **membind_bind**

enumerator **membind_interleave**

enumerator **membind_replicate**

enumerator **membind_nexttouch**

enumerator **membind_mixed**

enumerator **membind_user**

Functions

topology &**create_topology**()

inline *std*::size_t **get_memory_page_size**()

struct **hpx_hwloc_bitmap_wrapper**

Public Functions

HPX_NON_COPYABLE(*hpx_hwloc_bitmap_wrapper*)

inline **hpx_hwloc_bitmap_wrapper**()

inline **hpx_hwloc_bitmap_wrapper**(void *bmp)

inline ~**hpx_hwloc_bitmap_wrapper**()

inline void **reset**(hwloc_bitmap_t bmp)

```
inline explicit operator bool() const  
inline hwloc_bitmap_t get bmp() const
```

Private Members

```
hwloc_bitmap_t bmp_
```

Friends

```
friend std::ostream &operator<<(std::ostream &os, hpx_hwloc_bitmap_wrapper const *bmp)
```

```
struct topology
```

Public Functions

```
topology()
```

```
~topology()
```

```
inline 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.

```
inline 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.

inline 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

std::size_t get_cache_size(mask_type mask, int level) const
    Return the size of the cache associated with the given mask.

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 noexcept
    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)
inline std::size_t init_socket_number(std::size_t num_thread)

std::size_t init numa_node_number(std::size_t num_thread)
inline 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 get_number_of_core_pus_locked(std::size_t core) const
std::size_t extract_node_count(hwloc_obj_t parent, hwloc_obj_type_t type, std::size_t count)
                           const

std::size_t extract_node_count_locked(hwloc_obj_t parent, hwloc_obj_type_t type, std::size_t
                                         count) const

mask_type init_machine_affinity_mask() const
inline mask_type init_socket_affinity_mask(std::size_t num_thread) const
inline mask_type init numa_node_affinity_mask(std::size_t num_thread) const
inline mask_type init_core_affinity_mask(std::size_t num_thread) const
void init_num_of_pus()

hwloc_obj_t get_pu_obj(std::size_t num_core) const

```

Private Members

```
hwloc_topology_t topo

std::size_t num_of_pus_

bool use_pus_as_cores_

mutable 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

```
static mask_type empty_mask

static std::size_t memory_page_size_

static constexpr std::size_t pu_offset = 0

static constexpr std::size_t core_offset = 0
```

Friends

```
friend std::size_t get_memory_page_size()
```

util

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/util/insert_checked.hpp

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 Iterator>
inline 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.

Parameters **r** – [in] The return value of a std::map insert operation.

Returns This function returns **r.second**.

```
template<typename Iterator>
inline 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.

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**.

Returns This function returns **r.second**.

hpx/util/sed_transform.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **util**

Functions

```
bool parse_sed_expression(std::string const &input, std::string &search, std::string &replace)  
Parse a sed command.
```

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.
- **replace** – [out] If the parsing is successful, this string is set to the replace expression.

Returns *true* if the parsing was successful, false otherwise.

```
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
```

```
inline explicit operator bool() const noexcept
```

```
inline bool operator!() const
```

Private Members

```
std::shared_ptr<command> command_
```

actions

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/actions/action_support.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/actions/actions_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **actions**

hpx/actions/base_action.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/actions/transfer_action.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/actions/transfer_base_action.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

actions_base

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/actions_base/actions_base_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **actions**

hpx/actions_base/actions_base_support.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **actions**

namespace **hpx**

namespace **actions**

hpx/actions_base/basic_action.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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" << std::flush;
        }

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

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

Example:

Note: This macro has to be used once for each of the component actions defined using one of the `HPX_DEFINE_COMPONENT_ACTION` macros. It has to be visible in all translation units using the action, thus it is recommended to place it into the header file defining the component.

`HPX_REGISTER_ACTION_DECLARATION_(...)`

`HPX_REGISTER_ACTION_DECLARATION_1(action)`

`HPX_REGISTER_ACTION(...)`

Define the necessary component action boilerplate code.

The macro `HPX_REGISTER_ACTION` can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

Note: This macro has to be used once for each of the component actions defined using one of the `HPX_DEFINE_COMPONENT_ACTION` or `HPX_DEFINE_PLAIN_ACTION` macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note: Only one of the forms of this macro `HPX_REGISTER_ACTION` or `HPX_REGISTER_ACTION_ID` should be used for a particular action, never both.

`HPX_REGISTER_ACTION_ID(action, actionname, actionid)`

Define the necessary component action boilerplate code and assign a predefined unique id to the action.

The macro `HPX_REGISTER_ACTION` can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

The parameter *actionname* specifies an unique name of the action to be used for serialization purposes. The second parameter has to be usable as a plain (non-qualified) C++ identifier, it should not contain special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

The parameter *actionid* specifies an unique integer value which will be used to represent the action during serialization.

Note: This macro has to be used once for each of the component actions defined using one of the `HPX_DEFINE_COMPONENT_ACTION` or global actions `HPX_DEFINE_PLAIN_ACTION` macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note: Only one of the forms of this macro `HPX_REGISTER_ACTION` or `HPX_REGISTER_ACTION_ID` should be used for a particular action, never both.

namespace **hpx**

 namespace **actions**

hpx/actions_base/basic_action_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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

hpx/actions_base/component_action.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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>
    {
}
```

(continues on next page)

(continued from previous page)

```
void print_greeting() const
{
    hpx::cout << "Hey, how are you?\n" << std::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.

namespace **hpx**

namespace **actions**

hpx/actions_base/lambda_to_action.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/actions_base/plain_action.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

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;
    }
}
```

(continues on next page)

(continued from previous page)

```
// 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 **actions**

 namespace **traits**

hpx/actions_base/preassigned_action_id.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

 namespace **actions**

hpx/actions_base/traits/action_remote_result.hpp

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 Result>
using action_remote_result_t = typename action_remote_result<Result>::type
template<typename Result>
struct action_remote_result : public detail::action_remote_result_customization_point<Result>
```

agas

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/agas/addressing_service.hpp

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, *hpx*::id_type>

using **iterate_types_function_type** = *hpx*::function<void(*std*::string const&, *components*::component_type), true>

using **mutex_type** = *hpx*::spinlock

using **gva_cache_type** = *hpx*::util::cache::lru_cache<*gva_cache_key*, *gva*, *hpx*::util::cache::statistics::local_full_statistics>

using **migrated_objects_table_type** = *std*::set<*naming*::gid_type>

using **refcnt_requests_type** = *std*::map<*naming*::gid_type, *std*::int64_t>

using **resolved_localities_type** = *std*::map<*naming*::gid_type, *parcelset*::endpoints_type>

Public Functions

HPX_NON_COPYABLE(*addressing_service*)

explicit **addressing_service**(*util*::runtime_configuration const &*ini_*)

~addressing_service() = default

void bootstrap(*parcelset*::endpoints_type const &*endpoints*, *util*::runtime_configuration &*rtecfg*)

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.

inline state get_status() const

inline void set_status(state new_state)

inline 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)

inline bool is_bootstrap() const

inline bool is_console() const
    Returns whether this addressing_service represents the console locality.

inline 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, hpx::id_type const &id, std::int64_t compensated_credit)

inline server::primary_namespace &get_local_primary_namespace_service()

inline naming::address::address_type get_primary_ns_lva() const

inline naming::address::address_type get_symbol_ns_lva() const

inline server::component_namespace *get_local_component_namespace_service()

inline server::locality_namespace *get_local_locality_namespace_service()

inline server::symbol_namespace &get_local_symbol_namespace_service()

inline 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.

Returns 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.

Note: As long as *ec* is not pre-initialized to *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.

Returns This function returns *true* if a console locality_id exists and returns *false* otherwise.

```
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.

```
inline 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
```

```
inline 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.

Note: As long as *ec* is not pre-initialized to *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.

Returns The function returns the currently associated component type. Any error results in an exception thrown from this function.

```
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)
inline 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.

Note: As long as *ec* is not pre-initialized to `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.

Returns 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.

```
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.

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.

Returns 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.

```
inline 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.

Note: As long as *ec* is not pre-initialized to `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.

Returns This function returns *true*, if this global id got associated with an local address. It returns *false* otherwise.

```
inline hpx::future<bool> bind_async(naming::gid_type const &id, naming::address const &addr, std::uint32_t locality_id)
```

```
inline 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*).

Note: As long as *ec* is not pre-initialized to `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.

Returns This function returns *true*, if the given range was successfully bound. It returns *false* otherwise.

```
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)
```

```
inline 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)
```

```
inline bool unbind_local(naming::gid_type const &id, error_code &ec = throws)
```

Unbind a global address.

Remove the association of the given global address with any local address, which was bound to this global address. Additionally it returns the local address which was bound at the time of this call.

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.

Returns 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.

```
inline 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.

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.

Returns 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.

```
inline 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.

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.

Returns 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.

```
bool unbind_range_local(naming::gid_type const &lower_id, std::uint64_t count,
                        naming::address &addr, error_code &ec = throws)
```

Bind the given range of global ids.

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.

Returns 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.

```
hpx::future<naming::address> unbind_range_async(naming::gid_type const &lower_id,
                                                std::uint64_t count = 1)
```

inline 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.

Note: As long as *ec* is not pre-initialized to *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.

Returns This function returns *true* if the passed address refers to an object which lives on the locality of the caller.

```
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)
```

```
inline 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*).

Note: As long as *ec* is not pre-initialized to *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.

Returns 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.

```
inline bool resolve_local(hpx::id_type const &id, naming::address &addr, error_code &ec =  
throws)
```

```
inline naming::address resolve_local(naming::gid_type const &id, error_code &ec = throws)
```

```
inline naming::address resolve_local(hpx::id_type const &id, error_code &ec = throws)
```

```
hpx::future<naming::address> resolve_async(naming::gid_type const &id)
```

```
inline hpx::future<naming::address> resolve_async(hpx::id_type const &id)
```

```
hpx::future<hpx::id_type> get_colocation_id_async(hpx::id_type const &id)
```

```
bool resolve_full_local(naming::gid_type const &id, naming::address &addr, error_code &ec  
= throws)
```

```
inline bool resolve_full_local(hpx::id_type const &id, naming::address &addr, error_code &ec  
= throws)
```

```
inline naming::address resolve_full_local(naming::gid_type const &id, error_code &ec =  
throws)
```

```
inline naming::address resolve_full_local(hpx::id_type const &id, error_code &ec = throws)
```

```
hpx::future<naming::address> resolve_full_async(naming::gid_type const &id)
```

```
inline hpx::future<naming::address> resolve_full_async(hpx::id_type const &id)
```

```
bool resolve_cached(naming::gid_type const &id, naming::address &addr, error_code &ec =  
throws)
```

```
inline bool resolve_cached(hpx::id_type const &id, naming::address &addr, error_code &ec =  
throws)
```

```
inline bool resolve_local(naming::gid_type const *gids, naming::address *addrs, std::size_t size,
                           hpx::detail::dynamic_bitset<> &locals, error_code &ec = throws)
```

```
bool resolve_full_local(naming::gid_type const *gids, naming::address *addrs, std::size_t size,
                           hpx::detail::dynamic_bitset<> &locals, error_code &ec = throws)
```

```
bool resolve_cached(naming::gid_type const *gids, naming::address *addrs, std::size_t size,
                           hpx::detail::dynamic_bitset<> &locals, error_code &ec = throws)
```

```
hpx::future<std::int64_t> incref_async(naming::gid_type const &gid, std::int64_t credits = 1,
                                         hpx::id_type const &keep_alive = hpx::invalid_id)
```

Increment the global reference count for the given 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.

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.

Returns Whether the operation was successful.

```
inline 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.

Note: As long as *ec* is not pre-initialized to *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.

Returns The global reference count after the decrement.

```
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).

Note: As long as *ec* is not pre-initialized to *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.

Returns The function returns *true* if the global name was registered. It returns false if the global name is not registered.

```
hpx::future<bool> register_name_async(std::string const &name, hpx::id_type const &id)
```

```
bool register_name(std::string const &name, hpx::id_type const &id, error_code &ec = throws)
```

```
hpx::future<hpx::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).

Note: As long as *ec* is not pre-initialized to *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.

Returns 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.

```
hpx::id_type unregister_name(std::string const &name, error_code &ec = throws)
```

```
hpx::future<hpx::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.

Note: As long as *ec* is not pre-initialized to *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 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.

Returns [out] The id currently associated with the given global name (valid only if the return value is true).

```
hpx::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.

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`.

Returns A future instance encapsulating the global id which is causing the registered listener to be triggered.

```
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.

```
inline 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 &idbase, 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<hpx::id_type, naming::address>> **begin_migration**(*hpx::id_type const &id*)

start/stop migration of an object

Returns Current locality and address of the object to migrate

```
bool end_migration(hpx::id_type const &id)
```

std::pair<bool, components::pinned_ptr> **was_object_migrated**(*naming::gid_type const &gid,*

hpx::move_only_function<components::pinned_ptr> &&f)

Maintain list of migrated objects.

```
hpx::future<void> mark_as_migrated(naming::gid_type const &gid,
```

hpx::move_only_function<std::pair<bool,

hpx::future<void>>() &&f, bool

expect_to_be_marked_as_migrating)

Mark the given object as being migrated (if the object is unpinned). Delay migration until the object is unpinned otherwise.

```
void unmark_as_migrated(naming::gid_type const &gid)
```

Remove the given object from the table of migrated objects.

```
void pre_cache_endpoints(std::vector<parcels::endpoints_type> const&)
```

Public Members

mutable *mutex_type* **gva_cache_mtx_**

std::shared_ptr<gva_cache_type> **gva_cache_**

mutable *mutex_type* **migrated_objects_mtx_**

migrated_objects_table_type **migrated_objects_table_**

mutable *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_

mutable 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.
```

agas_base

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/agas_base/server/primary_namespace.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

Variables

```
HPX_ACTIONUSES_MEDIUMSTACK(hpx::agas::server::primary_namespace::allocate_action) HPX_REGISTER_ACTION
hpx::naming::address > std::pair<address_id_type
```

namespace **hpx**

namespace **agas**

Functions

naming::gid_type **bootstrap_primary_namespace_gid()**

hpx::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-----
BB
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.

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.
xxxxxxxxxxxxx000xxxxxxxxxxxxxx
Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxx
Prefix of the bootstrap AGAS locality.
000000010000000010000000000000000001
Address of the primary_namespace component on the bootstrap AGAS locality.
000000010000000010000000000000000002

(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.
```

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.

Variables

```
static constexpr char const *const primary_namespace_service_name = "primary/"

struct primary_namespace : public components::fixed_component_base<primary_namespace>
```

Public Types

```
using mutex_type = hpx::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

```
inline primary_namespace()  
  
void finalize()  
  
inline 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<hpx::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)  
  
hpx::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::detail::condition_variable>>  
  
using free_entry_allocator_type = util::internal_allocator<free_entry>  
  
using free_entry_list_type = std::list<free_entry, free_entry_allocator_type>
```

Private Functions

```
void wait_for_migration_locked(std::unique_lock<mutex_type> &l, naming::gid_type  
const &id, error_code &ec)  
  
resolved_type resolve_gid_locked(std::unique_lock<mutex_type> &l, naming::gid_type  
const &gid, error_code &ec)  
  
void increment(naming::gid_type const &lower, naming::gid_type const &upper, std::int64_t  
&credits, error_code &ec)  
  
void resolve_free_list(std::unique_lock<mutex_type> &l,  
                      std::list<refcnt_table_type::iterator> const &free_list,  
                      free_entry_list_type &free_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 gvas_  
  
refcnt_table_type refcnts_  
  
std::string instance_name_  
  
naming::gid_type next_id_  
  
naming::gid_type locality_  
  
migration_table_type migrating_objects_  
  
struct counter_data
```

Public Functions

```
HPX_NON_COPYABLE(counter_data)

counter_data() = default

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

inline **api_counter_data()**

Public Members

std::atomic<std::int64_t> **count_**

std::atomic<std::int64_t> **time_**

bool **enabled_**

struct **free_entry**

Public Functions

inline **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_**

async_colocated

See [Public API](#) for a list of names and headers that are part of the public HPX API.

`hpx/async_colocated/get_colocation_id.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

`hpx::id_type get_colocation_id(launch::sync_policy, hpx::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.

See also:

`hpx::get_colocation_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`.

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<hpx::id_type> get_colocation_id(hpx::id_type const &id)`

Asynchronously return the id of the locality where the object referenced by the given id is currently located on.

See also:

`hpx::get_colocation_id(launch::sync_policy)`

Parameters **id** – [in] The id of the object to locate.

async_distributed

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/async_distributed/base_lco.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

template<>

struct `hpx::get_lva<lcos::base_lco>`

Public Static Functions

static inline constexpr `lcos::base_lco *call(naming::address_type lva)` noexcept

template<>

struct `hpx::get_lva<lcos::base_lco> const`

Public Static Functions

static inline constexpr `lcos::base_lco const *call(naming::address_type lva)` noexcept

namespace `hpx`

template<> base_lco >

Public Static Functions

static inline constexpr `lcos::base_lco *call(naming::address_type lva)` noexcept

template<> base_lco const >

Public Static Functions

static inline constexpr `lcos::base_lco const *call(naming::address_type lva)` noexcept

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(hpx::id_type const&)`

`virtual void disconnect(hpx::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(hpx::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(hpx::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) HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco`

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,
connect_nonvirt, connect_action) HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco`

The `connect_action` may be used to.

The `set_exception_action` may be used to

Public Members

`set_exception_nonvirt`

`set_exception_action disconnect_nonvirt`

Public Static Functions

`static components::component_type get_component_type() noexcept`

`static void set_component_type(components::component_type type)`

hpx/async_distributed/base_lco_with_value.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

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, ActionIdSet)
```

```
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_6(Value, RemoteValue, Name, ActionIdGet, ActionIdSet, Tag)
```

namespace **hpx**

namespace **components**

namespace **lcos**

```
template<typename Result, typename RemoteResult, typename ComponentTagbase_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

```
using wrapping_type = typename detail::base_lco_wrapping_type<ComponentTag,
base_lco_with_value>::type
```

```
using base_type_holder = base_lco_with_value
```

Public Functions

```
inline 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.

```
inline 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) HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco_with_value
```

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

`get_value_nonvirt`

Public Static Functions

```
static inline components::component_type get_component_type() noexcept
```

```
static inline void set_component_type(components::component_type type)
```

Protected Types

```
using result_type = std::conditional_t<std::is_void_v<Result>, util::unused_type, Result>
```

Protected Functions

`~base_lco_with_value()` override = default

Destructor, needs to be virtual to allow for clean destruction of derived objects

```
inline virtual void set_event() override
```

```
inline void set_event_nonvirt(std::false_type)
```

```
inline void set_event_nonvirt(std::true_type)
```

```
virtual void set_value(RemoteResult &&result) = 0
```

```
virtual result_type get_value() = 0
```

```
inline 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

```
using wrapping_type = typename detail::base_lco_wrapping_type<ComponentTag,  
base_lco_with_value>::type
```

```
using base_type_holder = base_lco_with_value
```

```
using set_value_action = typename base_lco::set_event_action
```

Public Functions

```
inline void get_value()
```

Protected Functions

```
~base_lco_with_value() override = default
```

Destructor, needs to be virtual to allow for clean destruction of derived objects

```
namespace traits
```

hpx/async_distributed/lcos_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
namespace hpx
```

```
namespace distributed
```

```
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)  
hpx::distributed::promise<hpx::id_type> p;  
  
// Get the associated future  
future<hpx::id_type> f = p.get_future();  
  
// initiate the action supplying the promise as a
```

(continues on next page)

(continued from previous page)

```
// continuation
apply<some_action>(new continuation(p.get_id()), ...);

// Wait for the result to be returned, yielding control
// in the meantime.
hpx::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 **lcos**

TypeDefs

```
using instead = hpx::distributed::promise<Result, RemoteResult>

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>
template<typename ValueType>
struct object_semaphore

template<typename Action, typename Result = typename traits::promise_local_result<typename Action::remote_result_type>::type, bool DirectExecute = Action::direct_execution::value>
class packaged_action
    #include <packaged_action.hpp> A packaged_action can be used by a single thread to invoke a (remote) action and wait for the result. The result is expected to be sent back to the packaged_action using the LCO's set_event action
```

A *packaged_action* is one of the simplest synchronization primitives provided by HPX. It allows to synchronize on a eager evaluated remote operation returning a result of the type *Result*.

Note: The action executed using the *packaged_action* as a continuation must return a value of a type convertible to the type as specified by the template parameter *Result*.

Template Parameters

- **Action** – The template parameter *Action* defines the action to be executed by this *packaged_action* instance. The arguments *arg0, ..., argN* are used as parameters for this action.

- **Result** – The template parameter *Result* defines the type this *packaged_action* is expected to return from its associated future *packaged_action::get_future*.
- **DirectExecute** – The template parameter *DirectExecute* is an optimization aid allowing to execute the action directly if the target is local (without spawning a new thread for this). This template does not have to be supplied explicitly as it is derived from the template parameter *Action*.

```
namespace server

template<typename ValueType>
struct object_semaphore
```

namespace **lcos**

hpx/async_distributed/packaged_action.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **lcos**

```
template<typename Action, typename Result = typename traits::promise_local_result<typename Action::remote_result_type>::type, bool DirectExecute = Action::direct_execution::value>
class packaged_action
{
    #include <packaged_action.hpp>

    template<typename Action, typename Result>
    class packaged_action<Action, Result, false> : public hpx::distributed::promise<Result, hpx::traits::extract_action<Action>::remote_result_type>
        Subclassed by hpx::lcos::packaged_action<Action, Result, true >
```

Public Functions

```
inline packaged_action()

template<typename Allocatorpackaged_action(std::allocator_arg_t, Allocator const &alloc)

template<typename ...Ts>
inline void apply(hpx::id_type const &id, Ts&&... vs)

template<typename ...Ts>
inline void apply(naming::address &&addr, hpx::id_type const &id, Ts&&... vs)

template<typename Callback, typename ...Ts>
inline void apply_cb(hpx::id_type const &id, Callback &&cb, Ts&&... vs)

template<typename Callback, typename ...Ts>
```

```
inline void apply_cb(naming::address &&addr, hpx::id_type const &id, Callback &&cb, Ts&&... vs)

template<typename ...Ts>
inline void apply_p(hpx::id_type const &id, threads::thread_priority priority, Ts&&... vs)

template<typename ...Ts>
inline void apply_p(naming::address &&addr, hpx::id_type const &id, threads::thread_priority
priority, Ts&&... vs)

template<typename Callback, typename ...Ts>
inline void apply_p_cb(hpx::id_type const &id, threads::thread_priority priority, Callback &&cb,
Ts&&... vs)

template<typename Callback, typename ...Ts>
inline void apply_p_cb(naming::address &&addr, hpx::id_type const &id, threads::thread_priority
priority, Callback &&cb, Ts&&... vs)

template<typename ...Ts>
inline void apply_deferred(naming::address &&addr, hpx::id_type const &id, Ts&&... vs)

template<typename Callback, typename ...Ts>
inline void apply_deferred_cb(naming::address &&addr, hpx::id_type const &id, Callback
&&cb, Ts&&... vs)
```

Protected Types

```
using action_type = typename hpx::traits::extract_action<Action>::type

using remote_result_type = typename action_type::remote_result_type

using base_type = hpx::distributed::promise<Result, remote_result_type>
```

Protected Functions

```
template<typename ...Ts>
inline void do_apply(naming::address &&addr, hpx::id_type const &id, threads::thread_priority
priority, Ts&&... vs)

template<typename ...Ts>
inline void do_apply(hpx::id_type const &id, threads::thread_priority priority, Ts&&... vs)

template<typename Callback, typename ...Ts>
inline void do_apply_cb(naming::address &&addr, hpx::id_type const &id,
threads::thread_priority priority, Callback &&cb, Ts&&... vs)

template<typename Callback, typename ...Ts>
inline void do_apply_cb(hpx::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

inline **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**>

inline **packaged_action**(*std*::allocator_arg_t, *Allocator* const &alloc)

template<typename ...Ts>

inline void **apply**(*hpx*::id_type const &id, *Ts*&&... vs)

template<typename ...Ts>

inline void **apply**(*naming*::address &&addr, *hpx*::id_type const &id, *Ts*&&... vs)

template<typename **Callback**, typename ...Ts>

inline void **apply_cb**(*hpx*::id_type const &id, *Callback* &&cb, *Ts*&&... vs)

template<typename **Callback**, typename ...Ts>

inline void **apply_cb**(*naming*::address &&addr, *hpx*::id_type const &id, *Callback* &&cb, *Ts*&&... vs)

Private Types

using **action_type** = typename *packaged_action*<*Action*, *Result*, false>::action_type

`hpx/async_distributed/promise.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

template<>

```
class hpx::distributed::promise<void, hpx::util::unused_type> : public lcos::detail::promise_base<void, hpx::util::unused_type, lcos::detail::promise_data<void>>
```

Public Functions

promise() = default

constructs a promise object and a shared state.

template<typename **Allocator**>

inline **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) noexcept = default
    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() = default
    Abandons any shared state.

promise &operator=(promise &&other) noexcept = default
    Abandons any shared state (30.6.4) and then as if promise(HPX_MOVE(other)).swap(*this).

Returns *this.

inline void swap(promise &other) noexcept
    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.

inline void set_value()
    atomically stores the value r in the shared state and makes that state ready (30.6.4).

Throws 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

```
using base_type = lcos::detail::promise_base<void, hpx::util::unused_type,
lcos::detail::promise_data<void>>

template<typename R, typename Allocator>
struct uses_allocator<hpx::distributed::promise<R>, Allocator> : public true_type
    #include <promise.hpp> Requires: Allocator shall be an allocator (17.6.3.5)

namespace hpx

namespace distributed
```

Functions

```
template<typename Result, typename RemoteResult>
void swap(promise<Result, RemoteResult> &x, promise<Result, RemoteResult> &y) noexcept

template<typename Result, typename RemoteResult>
class promise
    #include <promise.hpp>

    template<> unused_type > : public lcos::detail::promise_base< void,
hpx::util::unused_type, lcos::detail::promise_data< void > >
```

Public Functions

promise() = default

constructs a promise object and a shared state.

template<typename **Allocator**>

inline **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) noexcept = default

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() = default

Abandons any shared state.

promise &operator=(promise &&other) noexcept = default

Abandons any shared state (30.6.4) and then as if *promise*(*HPX_MOVE*(other)).*swap*(*this).

Returns *this.

inline void **swap(promise &other)** noexcept

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.

inline void **set_value()**

atomically stores the value r in the shared state and makes that state ready (30.6.4).

Throws *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

```
using base_type = lcos::detail::promise_base<void, hpx::util::unused_type,
lcos::detail::promise_data<void>>
```

namespace **std**

```
template<typename R, typename Allocator> promise< R >,
Allocator > : public true_type
```

#include <*promise.hpp*> Requires: Allocator shall be an allocator (17.6.3.5)

hpx/async_distributed/transfer_continuation_action.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/async_distributed/trigger_lco.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

hpx/async_distributed/trigger_lco_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

void **trigger_lco_event**(*hpx::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*.

inline void **trigger_lco_event**(*hpx::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**(*hpx::id_type const &id*, *naming::address &&addr*, *hpx::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*.

```
inline void trigger_lco_event(hpx::id_type const &id, hpx::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(hpx::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(hpx::id_type const &id, Result &&t, bool move_credits = true)
```

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

Parameters

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

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

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(hpx::id_type const &id, naming::address &&addr, Result &&t, hpx::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(hpx::id_type
const
&id,
Re-
sult
&&t,
hpx::id_type
const
&cont,
bool
move_credits
=
true)
```

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

Parameters

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

```
template<typename Result>
```

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

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(hpx::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(hpx::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*.

```
inline void set_lco_error(hpx::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*.

inline void **set_lco_error**(*hpx*::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**(*hpx*::id_type const &*id*, *naming*::address &&*addr*, *std*::exception_ptr const &*e*,
hpx::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**(*hpx*::id_type const &*id*, *naming*::address &&*addr*, *std*::exception_ptr &&*e*,
hpx::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*.

inline void **set_lco_error**(*hpx*::id_type const &*id*, *std*::exception_ptr const &*e*, *hpx*::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*.

```
inline void set_lco_error(hpx::id_type const &id, std::exception_ptr &&e, hpx::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*.

checkpoint

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/checkpoint/checkpoint.hpp

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

```
inline 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.

Returns Operator<< returns the ostream object.

```
inline 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.

Returns Operator`>>` returns the ostream object.

```
template<typename T, typename ...Ts, typename U = typename
std::enable_if<!hpx::traits::is_launch_policy<T>::value && !std::is_same<typename
std::decay<T>::type, checkpoint>::value>::type>
hpx::future<checkpoint> save_checkpoint(T &&t, Ts&&... ts)
```

Save_checkpoint

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.

Returns 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.

Returns Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a checkpoint.

```
template<typename T, typename ...Ts, typename U = typename std::enable_if<!std::is_same<typename std::decay<T>::type, checkpoint>::value>::type>
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.

Returns 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.

Returns Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a

checkpoint.

```
template<typename T, typename ...Ts, typename U = typename std::enable_if<!std::is_same<typename std::decay<T>::type, checkpoint>::value>::type>
checkpoint save_checkpoint(hpx::launch::sync_policy sync_p, T &&t, Ts&&... ts)
Save_checkpoint - Sync_policy overload
```

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- **T** – Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **Ts** – More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **U** – This parameter is used to make sure that T is not a checkpoint. This forces the compiler to choose the correct overload.

Parameters

- **sync_p** – hpx::launch::sync_policy
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns 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.

Returns 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 && !std::is_same<typename std::decay<T>::type, checkpoint>::value>::type>
```

`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.

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.

Returns `prepare_checkpoint` returns a properly resized checkpoint object that can be used for a subsequent `restore_checkpoint` operation.

`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.

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.

Returns `prepare_checkpoint` returns a properly resized checkpoint object that can be used for a subsequent `restore_checkpoint` operation.

`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.

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.

Returns `prepare_checkpoint` returns a properly resized checkpoint object that can be used for a subsequent `restore_checkpoint` operation.

```
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.

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.

Returns prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

```
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.

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.

Returns Restore_checkpoint returns void.

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<char>::const_iterator
```

Public Functions

```
checkpoint() = default
~checkpoint() = default
checkpoint(checkpoint const &c) = default
checkpoint(checkpoint &&c) noexcept = default
inline checkpoint(std::vector<char> const &vec)
inline checkpoint(std::vector<char> &&vec) noexcept
checkpoint &operator=(checkpoint const &c) = default
checkpoint &operator=(checkpoint &&c) noexcept = default
inline const_iterator begin() const noexcept
inline const_iterator end() const noexcept
inline std::size_t size() const noexcept
inline char *data() noexcept
inline char const *data() const noexcept
```

Private Functions

```
template<typename Archive>
inline void serialize(Archive &arch, const unsigned int)
```

Private Members

```
std::vector<char> data_
```

Friends

```
friend class hpx::serialization::access
friend 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.

Returns Operator<< returns the ostream object.

```
friend 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.

Returns Operator>> returns the ostream object.

```
template<typename T, typename ...Ts>
```

```
friend 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.

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.

Returns Restore_checkpoint returns void.

```
inline friend bool operator==(checkpoint const &lhs, checkpoint const &rhs)
```

```
inline friend 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.

hpx/checkpoint_base/checkpoint_data.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **serialization**

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

```
struct checkpointing_tag
```

collectives

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/collectives/all_gather.hpp

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<decay_t<T>>> all_gather(char const *basename, T &&result,
                                                     num_sites_arg num_sites = num_sites_arg(),
                                                     this_site_arg this_site = this_site_arg(),
                                                     generation_arg generation = generation_arg(),
                                                     root_site_arg root_site = root_site_arg())
```

AllGather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

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).
- **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.
- **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. The generation number (if given) must be a positive number greater than zero. \params root_site The site that is responsible for creating the all_gather support object. This value is optional and defaults to ‘0’ (zero).

Returns 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.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> all_gather(communicator comm, T &&result,
                                                       this_site_arg this_site = this_site_arg(),
                                                       generation_arg generation =
                                                       generation_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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T>
hpx::future<std::vector<std::decay_t<T>>> all_gather(communicator comm, T &&result,
                                         generation_arg generation, 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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

hpx/collectives/all_reduce.hpp

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, 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.

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. The generation number (if given) must be a positive number greater than zero.
- **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).

Returns 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.

```
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(), generation_arg generation =
                                           generation_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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> all_reduce(communicator comm, T &&result, F &&op,
                                           generation_arg generation, 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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

hpx/collectives/all_to_all.hpp

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_to_all(char const *basename, T &&result,
                                                       num_sites_arg num_sites = num_sites_arg(),
                                                       this_site_arg this_site = this_site_arg(),
                                                       generation_arg generation = generation_arg(),
                                                       root_site_arg root_site = root_site_arg())
```

AllToAll a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

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. The generation number (if given) must be a positive number greater than zero.
- **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).

Returns 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.

template<typename T>

```
hpx::future<std::vector<std::decay_t<T>>> all_to_all(communicator comm, T &&result,  
this_site_arg this_site = this_site_arg(),  
generation_arg generation =  
generation_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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T>  
hpx::future<std::vector<std::decay_t<T>>> all_to_all(communicator comm, T &&result,  
generation_arg generation, 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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

hpx/collectives/argument_types.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **collectives**

```
struct generation_arg
```

Public Functions

```
inline explicit constexpr generation_arg(std::size_t generation = std::size_t(-1)) noexcept  
inline constexpr generation_arg &operator=(std::size_t generation) noexcept  
inline constexpr operator std::size_t() const noexcept
```

Public Members

std::size_t **generation_**

struct **num_sites_arg**

Public Functions

```
inline explicit constexpr num_sites_arg(std::size_t num_sites = std::size_t(-1)) noexcept  
inline constexpr num_sites_arg &operator=(std::size_t num_sites) noexcept  
inline constexpr operator std::size_t() const noexcept
```

Public Members

std::size_t **num_sites_**

struct **root_site_arg**

Public Functions

```
inline explicit constexpr root_site_arg(std::size_t root_site = std::size_t(0)) noexcept  
inline constexpr root_site_arg &operator=(std::size_t root_site) noexcept  
inline constexpr operator std::size_t() const noexcept
```

Public Members

std::size_t **root_site_**

struct **tag_arg**

Public Functions

```
inline explicit constexpr tag_arg(std::size_t tag = std::size_t(0)) noexcept  
inline constexpr tag_arg &operator=(std::size_t tag) noexcept  
inline constexpr operator std::size_t() const noexcept
```

Public Members

```
std::size_t tag_
```

```
struct that_site_arg
```

Public Functions

```
inline explicit constexpr that_site_arg(std::size_t that_site = std::size_t(-1)) noexcept  
inline constexpr that_site_arg &operator=(std::size_t that_site) noexcept  
inline constexpr operator std::size_t() const noexcept
```

Public Members

```
std::size_t that_site_
```

```
struct this_site_arg
```

Public Functions

```
inline explicit constexpr this_site_arg(std::size_t this_site = std::size_t(-1)) noexcept  
inline constexpr this_site_arg &operator=(std::size_t this_site) noexcept  
inline constexpr operator std::size_t() const noexcept
```

Public Members

```
std::size_t this_site_
```

hpx/collectives/barrier.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **distributed**

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 also:

`hpx::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

Returns 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

Returns 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.

hpx/collectives/broadcast.hpp

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<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 = generation_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.

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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns This function returns a future that will become ready once the broadcast operation has been completed.

```
template<typename T>
hpx::future<void> broadcast_to(communicator comm, T &&local_result, this_site_arg this_site =
    this_site_arg(), generation_arg generation = generation_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.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future that will become ready once the broadcast operation has been completed.

```
template<typename T>
hpx::future<void> broadcast_to(communicator comm, generation_arg generation, 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.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – A value to transmit to all participating sites from this call site.
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns This function returns a future that will become ready once the broadcast operation has been completed.

```
template<typename T>
hpx::future<T> broadcast_from(char const *basename, this_site_arg this_site = this_site_arg(),
    generation_arg generation = generation_arg())
```

Receive a value that was broadcast to different call sites

This function sends a set of values to all call sites operating on the given base name.

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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

```
template<typename T>
hpx::future<T> broadcast_from(communicator comm, this_site_arg this_site = this_site_arg(),
                               generation_arg generation = generation_arg())
```

Receive a value that was broadcast to different call sites

This function sends a set of values to all call sites operating on the given base name.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T>
hpx::future<T> broadcast_from(communicator comm, generation_arg generation, 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.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

[hpx/collectives/broadcast_direct.hpp](#)

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

Perform a distributed broadcast operation.

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

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

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.

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

```
template<typename Action, typename ArgN, ...
> void broadcast_apply (std::vector< hpx::id_type > const &ids, ArgN argN, ...)
```

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

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

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

Parameters

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

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

Perform a distributed broadcast operation.

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

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

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

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.

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

```
template<typename Action, typename ArgN, ...>
> void broadcast_apply_with_index (std::vector< hpx::id_type > const &ids, ArgN argN, ...)
```

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

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

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

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

Parameters

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

`hpx/collectives/channel_communicator.hpp`

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

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.

Returns This function returns a future to a new communicator object usable with the collective operation.

```
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.

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.

Returns This function returns a new communicator object usable with the collective operation.

```
template<typename T>
hpx::future<void> set(channel_communicator comm, that_site_arg site, T &&value, tag_arg tag =
                        tag_arg())
```

Send a value to the given site

This function sends a value to the given site based on the given communicator.

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

Returns This function returns a future<void> that becomes ready once the data transfer operation has finished.

```
template<typename T>
hpx::future<T> get(channel_communicator comm, that_site_arg site, tag_arg tag = tag_arg())
```

Send a value to the given site

This function receives a value from the given site based on the given communicator.

Parameters

- **comm** – The channel communicator object to use for the data transfer
- **site** – The source site

Returns This function returns a future<T> that becomes ready once the data transfer operation has finished. The future will hold the received value.

hpx/collectives/communication_set.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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).

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)

Returns This function returns a future holding an id_type of the communicator object to be used on the current locality.

hpx/collectives/create_communicator.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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 functions 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.).

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).

Returns This function returns a new communicator object usable with the collective operation.

`hpx/collectives/exclusive_scan.hpp`

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, 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 *root_site*.

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).
- **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.
- **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 supplied only if the exclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero. \params root_site The site that is responsible for creating the exclusive_scan support object. This value is optional and defaults to '0' (zero).

Returns 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.

```
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(),
                                                 generation_arg generation = generation_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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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 supplied only if the exclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> exclusive_scan(communicator comm, T &&result, F &&op,
                                                 generation_arg generation, 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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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
- **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 supplied only if the exclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

hpx/collectives/fold.hpp

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

Perform a distributed fold operation.

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

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

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.

Returns This function returns a future representing the result of the overall folding operation.

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

Perform a distributed folding operation.

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

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

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

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.

Returns This function returns a future representing the result of the overall folding operation.

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

Perform a distributed inverse folding operation.

The function `hpx::lcos::inverse_fold` performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action).

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

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.

Returns This function returns a future representing the result of the overall folding operation.

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

Perform a distributed inverse folding operation.

The function `hpx::lcos::inverse_fold_with_index` performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action).

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

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

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.

Returns This function returns a future representing the result of the overall folding operation.

hpx/collectives/gather.hpp

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<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.

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).

- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_here(communicator comm, T &&result, 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.

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_here(communicator comm, T &&result,
                                                       generation_arg generation, 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.

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

```
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)

Parameters

- **basename** – The base name identifying the gather operation
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.
- **root_site** – The sequence number of the central gather point (usually the locality id). This value is optional and defaults to 0.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_there(communicator comm, T &&result, this_site_arg
                                                       this_site = this_site_arg(), generation_arg
                                                       generation = generation_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)

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_there(communicator comm, T &&result,
                                                       generation_arg generation, 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)

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

hpx/collectives/inclusive_scan.hpp

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, 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.

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).
- **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.
- **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. The generation number (if given) must be a positive number greater than zero. \params root_site The site that is responsible for creating the inclusive_scan support object. This value is optional and defaults to ‘0’ (zero).

Returns 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.

```
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(),
                                                generation_arg generation = generation_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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> inclusive_scan(communicator comm, T &&result, F &&op,
                                                generation_arg generation, 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.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

hpx/collectives/latch.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **distributed**

```
class latch : public components::client_base<latch, hpx::lcos::server::latch>
```

Public Functions

latch() = default

explicit **latch**(*std*::ptrdiff_t count)

Initialize the latch

Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

inline **latch**(*hpx*::id_type const &id)

Extension: Create a client side representation for the existing *server*::*latch* instance with the given global id *id*.

inline **latch**(*hpx*::*future*<*hpx*::id_type> &&f)

Extension: Create a client side representation for the existing *server*::*latch* instance with the given global id *id*.

inline **latch**(*hpx*::*shared_future*<*hpx*::id_type> const &id)

Extension: Create a client side representation for the existing *server*::*latch* instance with the given global id *id*.

inline **latch**(*hpx*::*shared_future*<*hpx*::id_type> &&id)

inline 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.

Throws Nothing. –

inline void **arrive_and_wait**()

Decrements counter_ by update . 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.

Throws Nothing. –

inline 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 .

Throws Nothing. –

inline bool **is_ready**() const noexcept

Returns: counter_ == 0. Does not block.

Throws Nothing. –

inline bool **try_wait**() const noexcept

Returns: counter_ == 0. Does not block.

Throws Nothing. –

```
inline void wait() const
    If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization
    point until counter_ reaches 0.
    Throws Nothing.
```

Private Types

`typedef components::client_base<latch, hpx::lcos::server::latch> base_type`

namespace **lcos**

hpx/collectives/reduce.hpp

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, 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.

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).
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
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(), generation_arg
                                         generation = generation_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.

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T, typename F>
hpx::future<decay_t<T>> reduce_here(communicator comm, T &&local_result, F &&op,
                                         generation_arg generation, 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.

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

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
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

```
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)

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.
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.
- **root_site** – The sequence number of the central reduction point (usually the locality id). This value is optional and defaults to 0.

Returns This function returns a future<void>. It will become ready once the reduction operation has been completed.

```
template<typename T>
hpx::future<void> reduce_there(communicator comm, T &&local_result, this_site_arg this_site =
    this_site_arg(), generation_arg generation = generation_arg())
```

Reduce a given value at the given call site

This function transmits the value given by *result* to a central reduce site (where the corresponding *reduce_here* is executed)

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns 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.

```
template<typename T>
hpx::future<void> reduce_there(communicator comm, T &&local_result, generation_arg generation,
    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)

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – A value to reduce on the central reduction point from this call site.
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns 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.

[hpx/collectives/reduce_direct.hpp](#)

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 Action, typename ReduceOp, typename ArgN, ...
> hpx::future< decltype(Action(hpx::id_type, ArgN, ...
))> reduce (std::vector< hpx::id_type > const &ids, ReduceOp &&reduce_op,
ArgN argN,...)
```

Perform a distributed reduction operation.

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

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.

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

```
template<typename Action, typename ReduceOp, typename ArgN, ...
> hpx::future< decltype(Action(hpx::id_type, ArgN,...,
std::size_t))> reduce_with_index (std::vector< hpx::id_type > const &ids,
ReduceOp &&reduce_op, ArgN argN,...)
```

Perform a distributed reduction operation.

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

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

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.

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

hpx/collectives/scatter.hpp

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<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.

Parameters

- **basename** – The base name identifying the scatter operation
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.
- **root_site** – The sequence number of the central scatter point (usually the locality id). This value is optional and defaults to 0.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

```
template<typename T>
hpx::future<T> scatter_from(communicator comm, this_site_arg this_site = this_site_arg(),
                             generation_arg generation = generation_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.

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

```
template<typename T>
hpx::future<T> scatter_from(communicator comm, generation_arg generation, 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.

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

```
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)

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 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. The generation number (if given) must be a positive number greater than zero.
- **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.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

```
template<typename T>
hpx::future<T> scatter_to(communicator comm, std::vector<T> &&result, 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)

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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.
- **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. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

```
template<typename T>
hpx::future<T> scatter_to(communicator comm, std::vector<T> &&result, generation_arg
                           generation, 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)

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **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. The generation number (if given) must be a positive number greater than zero.

- **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.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

components

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/components/basename_registration.hpp

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.

This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

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.

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.

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.
- **base_name** – [in] The base name for which to retrieve the registered ids.
- **num_ids** – [in] The number of registered ids to expect.

Returns A list of futures representing the ids which were registered using the given base name.

Returns A list of futures representing the ids which were registered using the given base name.

```
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.

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.

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.

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.

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.
- `base_name` – [in] The base name for which to retrieve the registered ids.
- `ids` – [in] The sequence numbers of the registered ids.

Returns A list of futures representing the ids which were registered using the given base name and sequence numbers.

Returns A list of futures representing the ids which were registered using the given base name and sequence numbers.

```
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.

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.

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.

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.
- **base_name** – [in] The base name for which to retrieve the registered ids.
- **sequence_nr** – [in] The sequence number of the registered id.

Returns A representing the id which was registered using the given base name and sequence numbers.

Returns A representing the id which was registered using the given base name and sequence numbers.

```
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.

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.

Returns A future representing the result of the registration operation itself.

```
template<typename Client>
Client unregister_with_basename(std::string base_name, std::size_t sequence_nr =
                                ~static_cast<std::size_t>(0))
```

Unregister the given id using the given base name.

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.

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.
- **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.

Returns A future representing the result of the un-registration operation itself.

Returns A future representing the result of the un-registration operation itself.

hpx/components/basename_registration_fwd.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

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.

Returns A future representing the result of the registration operation itself.

```
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.

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.

Returns A future representing the result of the registration operation itself.

hpx/components/components_fwd.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

```
template<typename Derived, typename Stub>
```

```
class client_base
```

namespace **components**

hpx/components/get_ptr.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename Component>
hpx::future<std::shared_ptr<Component>> get_ptr(hpx::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.

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.

Returns This function returns a future representing the pointer to the underlying memory for the component instance with the given `id`.

```
template<typename Derived, typename Stub>
hpx::future<std::shared_ptr<typename components::client_base<Derived, Stub>::server_component_type>> get_ptr(components::client_base<Derived, Stub>::server_component_type const &c)
```

Returns a future referring to the pointer to the underlying memory of a component.

The function `hpx::get_ptr` can be used to extract a future referring to the pointer to the underlying memory of a given component.

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.

Returns This function returns a future representing the pointer to the underlying memory for the component instance with the given `id`.

```
template<typename Component>
std::shared_ptr<Component> get_ptr(launch::sync_policy p, hpx::id_type const &id, error_code &ec = throws)
```

Returns the pointer to the underlying memory of a component.

The function `hpx::get_ptr_sync` can be used to extract the pointer to the underlying memory of a given component.

Note: This function will successfully return the requested result only if the given component is currently located on the requesting locality. Otherwise the function will raise an error.

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

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

Parameters

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

Template Parameters **The** – only template parameter has to be the type of the server side component.

Returns This function returns the pointer to the underlying memory for the component instance with the given `id`.

```
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<Der-
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.

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.

Returns This function returns the pointer to the underlying memory for the component instance with the given *id*.

components_base

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/components_base/agas_interface.hpp

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, hpx::id_type const &id, error_code
                    &ec = throws)

hpx::future<bool> register_name(std::string const &name, hpx::id_type const &id)

hpx::id_type unregister_name(launch::sync_policy, std::string const &name, error_code &ec =
                                throws)

hpx::future<hpx::id_type> unregister_name(std::string const &name)
```

```
hpx::id_type resolve_name(launch::sync_policy, std::string const &name, error_code &ec = throws)  
hpx::future<hpx::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)  
inline 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)  
inline hpx::naming::gid_type get_locality()  
std::vector<std::uint32_t> get_all_locality_ids(naming::component_type type, error_code &ec =  
                                                throws)  
inline 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)  
inline bool is_local_address_cached(hpx::id_type const &id, error_code &ec = throws)  
inline bool is_local_address_cached(hpx::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)  
inline bool is_local_lva_encoded_address(hpx::id_type const &id)  
hpx::future<naming::address> resolve(hpx::id_type const &id)  
naming::address resolve(launch::sync_policy, hpx::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(hpx::id_type const &id, error_code &ec = throws)
    Invoke an asynchronous garbage collection step on the given target locality.

void garbage_collect(hpx::id_type const &id, error_code &ec = throws)
    Invoke a synchronous garbage collection step on the given target locality.

hpx::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, hpx::id_type const
                                    &keep_alive = hpx::invalid_id)

std::int64_t incref(launch::sync_policy, naming::gid_type const &gid, std::int64_t credits = 1,
                        hpx::id_type const &keep_alive = hpx::invalid_id, error_code &ec = throws)

std::int64_t replenish_credits(naming::gid_type &gid)

hpx::future<hpx::id_type> get_colocation_id(hpx::id_type const &id)

hpx::id_type get_colocation_id(launch::sync_policy, hpx::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<hpx::id_type, naming::address>> begin_migration(hpx::id_type const &id)

```

```
bool end_migration(hpx::id_type const &id)
hpx::future<void> mark_as_migrated(naming::gid_type const &gid,
hpx::move_only_function<std::pair<bool,
hpx::future<void>>() &&f, bool
expect_to_be_marked_as_migrating)
```

```
std::pair<bool, components::pinned_ptr> was_object_migrated(naming::gid_type const &gid,
hpx::move_only_function<components::pinned_ptr()>
&&f)
```

```
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()
```

```
struct agas_interface_functions &agas_init()
```

hpx/components_base/component_commandline.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 implementation of a component's startup/shutdown function provider.
```

Public Functions

inline `hpx::program_options::options_description add_commandline_options()` override
 Return any additional command line options valid for this component.

Note: This function will be executed by the runtime system during system startup.

Returns The module is expected to fill a `options_description` object with any additional command line options this component will handle.

namespace `commandline_options_provider`

Functions

`hpx::program_options::options_description add_commandline_options()`

`hpx/components_base/component_startup_shutdown.hpp`

See *Public API* for a list of names and headers that are part of the public HPX API.

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)(startup_function_type&, bool&), bool
        (*Shutdown)(shutdown_function_type&, bool&)>
struct component_startup_shutdown : public component_startup_shutdown_base
```

`#include <component_startup_shutdown.hpp>` The `component_startup_shutdown` class provides a minimal implementation of a component's startup/shutdown function provider.

Public Functions

```
inline bool get_startup_function(startup_function_type &startup, bool &pre_startup) override
```

Return any startup function for this component.

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.

Returns Returns *true* if the parameter *startup* has been successfully initialized with the startup function.

```
inline bool get_shutdown_function(shutdown_function_type &shutdown, bool &pre_shutdown) override
```

Return any shutdown function for this component.

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.

Returns Returns *true* if the parameter *shutdown* has been successfully initialized with the shutdown function.

hpx/components_base/component_type.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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:

```
enumerator component_invalid
```

```
enumerator component_runtime_support
```

```
enumerator component_plain_function
```

```
enumerator component_base_lco
```

```
enumerator component_base_lco_with_value_unmanaged
```

```
enumerator component_base_lco_with_value
```

```
enumerator component_latch
```

```
enumerator component_barrier
```

```
enumerator component.promise
```

```
enumerator component_agas_locality_namespace
```

```
enumerator component_agas_primary_namespace
```

```
enumerator component_agas_component_namespace
```

```
enumerator component_agas_symbol_namespace
```

```
enumerator component_last
```

```
enumerator component_first_dynamic
```

```
enumerator component_upper_bound
```

```
enum factory_state_enum
```

Values:

```
enumerator factory_enabled
```

```
enumerator factory_disabled
```

```
enumerator factory_check
```

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(hpx::move_only_function<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.

```
inline constexpr component_type get_base_type(component_type t) noexcept
```

The lower short word of the component type is the type of the component exposing the actions.

```
inline constexpr component_type get_derived_type(component_type t) noexcept
```

The upper short word of the component is the actual component type.

```
inline constexpr component_type derived_component_type(component_type derived,  
component_type base) noexcept
```

A component derived from a base component exposing the actions needs to have a specially formatted component type.

```
inline constexpr bool types_are_compatible(component_type lhs, component_type rhs) noexcept
```

Verify the two given component types are matching (compatible)

```
template<typename Component, typename Enable = void>  
char const *get_component_name() noexcept
```

```
template<typename Component, typename Enable = void>  
const char *get_component_base_name() noexcept
```

```
template<typename Component>  
inline component_type get_component_type() noexcept
```

```
template<typename Component>  
inline void set_component_type(component_type type)
```

namespace **naming**

Functions

```
std::ostream &operator<<(std::ostream&, address const&)
```

hpx/components_base/components_base_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

TypeDefs

```
using instead = abstract_component_base<Component>

template<typename Component = detail::this_type>
class abstract_component_base

template<typename Component, typename Derived = detail::this_type>
class abstract_managed_component_base

template<typename Component>
class component

template<typename Component = detail::this_type>
class component_base

template<typename Component>
class fixed_component

template<typename Component>
class fixed_component_base

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** –

```
template<typename Component, typename Wrapper, typename CtorPolicy, typename DtorPolicy>
class managed_component_base
```

namespace **components**

hpx/components_base/get_lva.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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 inline constexpr Component *call(naming::address_type lva) noexcept
```

hpx/components_base/server/fixed_component_base.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

```
template<typename Component>
```

```
class fixed_component
```

```
template<typename Component>
```

```
class fixed_component_base
```

`hpx/components_base/server/managed_component_base.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

```
template<>

struct hpx::components::detail_adl_barrier::init<traits::construct_with_back_ptr>
```

Public Static Functions

```
template<typename Component, typename Managed>
static inline constexpr void call(Component*, Managed*) noexcept

template<typename Component, typename Managed, typename ...Ts>
static inline void call_new(Component *component, Managed *this_, Ts&&... vs)
```

```
template<>

struct hpx::components::detail_adl_barrier::init<traits::construct_without_back_ptr>
```

Public Static Functions

```
template<typename Component, typename Managed>
static inline void call(Component *component, Managed *this_)

template<typename Component, typename Managed, typename ...Ts>
static inline void call_new(Component *component, Managed *this_, Ts&&... vs)
```

```
template<>

struct
hpx::components::detail_adl_barrier::destroy_backptr<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```
template<typename BackPtr>
static inline void call(BackPtr *back_ptr)
```

```
template<>

struct
hpx::components::detail_adl_barrier::destroy_backptr<traits::managed_object_controls_lifetime>
```

Public Static Functions

```
template<typename BackPtr>
static inline constexpr void call(BackPtr*) noexcept
```

```
template<>

struct
hpx::components::detail_adl_barrier::manage_lifetime<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```
template<typename Component>
static inline constexpr void call(Component*) noexcept

template<typename Component>
static inline void addref(Component *component) noexcept

template<typename Component>
static inline void release(Component *component) noexcept

template<>

struct
hpx::components::detail_adl_barrier::manage_lifetime<traits::managed_object_controls_lifetime>
```

Public Static Functions

```
template<typename Component>
static inline void call(Component *component) noexcept(noexcept(component->finalize()))

template<typename Component>
static inline constexpr void addref(Component*) noexcept

template<typename Component>
static inline constexpr void release(Component*) noexcept
```

namespace **hpx**

namespace **components**

Functions

```
template<typename Component, typename Derived>
void intrusive_ptr_add_ref(managed_component<Component, Derived> *p) noexcept

template<typename Component, typename Derived>
void intrusive_ptr_release(managed_component<Component, Derived> *p) noexcept

template<typename Component, typename Derived>
class managed_component
{
    #include <managed_component_base.hpp>
};

template<typename Component, typename Wrapper, typename CtorPolicy, typename DtorPolicy>
class managed_component_base

namespace detail_adl_barrier

template<typename DtorTag>
struct destroy_backptr
```

```
template<> managed_object_controls_lifetime >
```

Public Static Functions

```
template<typename BackPtr>
static inline constexpr void call(BackPtr*) noexcept
```

```
template<> managed_object_is_lifetime_controlled >
```

Public Static Functions

```
template<typename BackPtr>
static inline void call(BackPtr *back_ptr)
```

```
template<typename BackPtrTag>
```

```
struct init
```

```
template<> construct_with_back_ptr >
```

Public Static Functions

```
template<typename Component, typename Managed>
static inline constexpr void call(Component*, Managed*) noexcept
```

```
template<typename Component, typename Managed, typename ...Ts>
static inline void call_new(Component *&component, Managed *this_, Ts&&... vs)
```

```
template<> construct_without_back_ptr >
```

Public Static Functions

```
template<typename Component, typename Managed>
static inline void call(Component *component, Managed *this_)
```

```
template<typename Component, typename Managed, typename ...Ts>
static inline void call_new(Component *&component, Managed *this_, Ts&&... vs)
```

```
template<typename DtorTag>
```

```
struct manage_lifetime
```

```
template<> managed_object_controls_lifetime >
```

Public Static Functions

```
template<typename Component>
static inline void call(Component *component) noexcept(noexcept(component->finalize()))

template<typename Component>
static inline constexpr void addref(Component*) noexcept

template<typename Component>
static inline constexpr void release(Component*) noexcept

template<> managed_object_is_lifecycle_controlled >
```

Public Static Functions

```
template<typename Component>
static inline constexpr void call(Component*) noexcept

template<typename Component>
static inline void addref(Component *component) noexcept

template<typename Component>
static inline void release(Component *component) noexcept
```

compute

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/compute/host/target_distribution_policy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

distribution_policies

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/distribution_policies/binpacking_distribution_policy.hpp

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/component@"
```

```
static 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

```
inline binpacking_distribution_policy()
```

Default-construct a new instance of a *binpacking_distribution_policy*. This policy will represent one locality (the local locality).

```
inline binpacking_distribution_policy operator() (std::vector<id_type> const &locs, char const
                                                 *perf_counter_name =
                                                 default_binpacking_counter_name) const
```

Create a new *default_distribution* policy representing the given set of localities.

Parameters

- **locs** – [in] The list of localities the new instance should represent
- **perf_counter_name** – [in] The name of the performance counter which should be used as the distribution criteria (by default the overall number of existing instances of the given component type will be used).

```
inline binpacking_distribution_policy operator() (std::vector<id_type> &&locs, char const
                                                 *perf_counter_name =
                                                 default_binpacking_counter_name) const
```

Create a new *default_distribution* policy representing the given set of localities.

Parameters

- **locs** – [in] The list of localities the new instance should represent
- **perf_counter_name** – [in] The name of the performance counter which should be used as the distribution criteria (by default the overall number of existing instances of the given component type will be used).

```
inline binpacking_distribution_policy operator() (id_type const &loc, char const
                                                 *perf_counter_name =
                                                 default_binpacking_counter_name) const
```

Create a new *default_distribution* policy representing the given locality

Parameters

- **loc** – [in] The locality the new instance should represent
- **perf_counter_name** – [in] The name of the performance counter that should be used as the distribution criteria (by default the overall number of existing instances of the given component type will be used).

```
template<typename Component, typename ...Ts>
inline hpx::future<hpx::id_type> create(Ts&&... vs) const
    Create one object on one of the localities associated by this policy instance
    Parameters vs – [in] The arguments which will be forwarded to the constructor of the new
        object.
    Returns A future holding the global address which represents the newly created object

template<typename Component, typename ...Ts>
inline 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
    Parameters
        • count – [in] The number of objects to create
        • vs – [in] The arguments which will be forwarded to the constructors of the new objects.
    Returns A future holding the list of global addresses which represent the newly created
        objects

inline std::string const &get_counter_name() const
    Returns the name of the performance counter associated with this policy instance.

inline 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/distribution_policies/colocating_distribution_policy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

Variables

```
static 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

Default-construct a new instance of a `colocating_distribution_policy`. This policy will represent the local locality.

inline `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**>

inline `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**>

inline `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

Parameters `vs` – [in] The arguments which will be forwarded to the constructor of the new object.

Returns A future holding the global address which represents the newly created object

template<typename **Component**, typename ...**Ts**>

inline `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

Parameters

- `count` – [in] The number of objects to create
- `vs` – [in] The arguments which will be forwarded to the constructors of the new objects.

Returns A future holding the list of global addresses which represent the newly created objects

template<typename **Action**, typename ...**Ts**>

inline `async_result<Action>::type async(launch policy, Ts&&... vs)` const

template<typename **Action**, typename **Callback**, typename ...**Ts**>

inline `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**>

```
inline 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>
```

```
inline bool apply(threads::thread_priority priority, Ts&&... vs) const
```

```
template<typename Action, typename Continuation, typename Callback, typename ...Ts>
```

```
inline 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>
```

```
inline bool apply_cb(threads::thread_priority priority, Callback &&cb, Ts&&... vs) const
```

```
inline 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

```
inline 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

```
using type = hpx::future<typename traits::promise_local_result<typename  
                           hpx::traits::extract_action<Action>::remote_result_type>::type>
```

[hpx/distribution_policies/target_distribution_policy.hpp](#)

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

Variables

```
static 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
```

Default-construct a new instance of a *target_distribution_policy*. This policy will represent one locality (the local locality).

```
inline 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>
```

```
inline 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

Parameters **vs** – [in] The arguments which will be forwarded to the constructor of the new object.

Returns A future holding the global address which represents the newly created object

```
template<typename Component, typename ...Ts>
```

```
inline 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

Parameters

- **count** – [in] The number of objects to create
- **vs** – [in] The arguments which will be forwarded to the constructors of the new objects.

Returns A future holding the list of global addresses which represent the newly created objects

```
template<typename Action, typename ...Ts>
```

```
inline async_result<Action>::type async(launch policy, Ts&&... vs) const
```

```
template<typename Action, typename Callback, typename ...Ts>
```

```
inline 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>
inline 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>
inline bool apply(threads::thread_priority priority, Ts&&... vs) const
```

```
template<typename Action, typename Continuation, typename Callback, typename ...Ts>
inline 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>
inline bool apply_cb(threads::thread_priority priority, Callback &&cb, Ts&&... vs) const
```

```
inline 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

```
inline 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 <target_distribution_policy.hpp>
```

Note: This function is part of the invocation policy implemented by this class

Public Types

```
using type = hpx::future<typename traits::promise_local_result<typename
hpx::traits::extract_action<Action>::remote_result_type>::type>
```

[hpx/distribution_policies/unwrapping_result_policy.hpp](#)

See *Public API* for a list of names and headers that are part of the public HPX API.

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

```
inline explicit unwrapping_result_policy(id_type const &id)
```

```
template<typename Client, typename Stub>
```

```
inline explicit unwrapping_result_policy(client_base<Client, Stub> const &client)
```

```
template<typename Action, typename ...Ts>
```

```
inline async_result<Action>::type async(launch policy, Ts&&... vs) const
```

```
template<typename Action, typename ...Ts>
```

```
inline async_result<Action>::type async(launch::sync_policy, Ts&&... vs) const
```

```
template<typename Action, typename Callback, typename ...Ts>
```

```
inline async_result<Action>::type async_cb(launch policy, Callback &&cb, Ts&&... vs) const
```

```
template<typename Action, typename Continuation, typename ...Ts>
```

```
inline 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>
```

```
inline bool apply(threads::thread_priority priority, Ts&&... vs) const
```

```
template<typename Action, typename Continuation, typename Callback, typename ...Ts>
```

```
inline 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>
```

```
inline bool apply_cb(threads::thread_priority priority, Callback &&cb, Ts&&... vs) const
```

```
inline hpx::id_type const &get_next_target() const
```

```
template<typename Action>
```

```
struct async_result
```

Public Types

using **type** = typename *traits*::promise_local_result<typename *hpx::traits*::extract_action<*Action*>::remote_result_type>::type

executors_distributed

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/executors_distributed/distribution_policy_executor.hpp

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(DistPolicy const& policy);
```

Create a new *distribution_policy_executor* from the given distribution_policy.

Parameters `policy` – The distribution_policy to create an executor from

```
template<typename DistPolicy>
```

```
class distribution_policy_executor
```

```
#include <distribution_policy_executor.hpp> A distribution_policy_executor creates groups of parallel execution agents which execute in threads implicitly created by the executor and placed on any of the associated localities.
```

Template Parameters `DistPolicy` – The distribution policy type for which an executor should be created. The expression `hpx::traits::is_distribution_policy<DistPolicy>::value` must evaluate to true.

Public Functions

```
template<typename DistPolicy_, typename Enable =  
std::enable_if_t<!std::is_same_v<distribution_policy_executor, std::decay_t<DistPolicy_>>>>  
inline 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_**

init_runtime

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/hpx_finalize.hpp

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.

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.

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.

Using this function is an alternative to *hpx::disconnect*, these functions do not need to be called both.

Note: As long as *ec* is not pre-initialized to *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

- **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.
- **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.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will always return zero.

```
inline 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.

Using this function is an alternative to *hpx::disconnect*, these functions do not need to be called both.

Note: As long as *ec* is not pre-initialized to *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.

Returns This function will always return zero.

```
void 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.

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.

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.

Note: As long as `ec` is not pre-initialized to `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

- **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.
- **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.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to `hpx::throws` the function will throw on error instead.

Returns This function will always return zero.

inline 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.

Note: As long as `ec` is not pre-initialized to `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.

Returns This function will always return zero.

```
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`.

Returns The function returns the value, which has been returned from the user supplied main HPX function (usually `hpx_main`).

hpx/hpx_init.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

hpx/hpx_init_impl.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

Functions

```
inline 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.

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 `parametemode`.

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`)

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

```
inline 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.

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`)

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

```
inline 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.

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`)

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

```
inline 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.

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`)

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

```
inline 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).

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](#))

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

hpx/hpx_init_params.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX API*.

namespace `hpx`

```
struct init_params
#include <hpx_init_params.hpp> Parameters used to initialize the HPX runtime through hpx::init and hpx::start.
```

Public Functions

inline `init_params()`

Public Members

```
std::reference_wrapper<hpx::program_options::options_description const> desc_cmdline =
hpx::local::detail::default_desc(HPX_APPLICATION_STRING)
```

`std::vector<std::string> cfg`

mutable `startup_function_type startup`

mutable `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
```

hpx/hpx_start.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

hpx/hpx_start_impl.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
inline 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.

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`)

Returns The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

```
inline 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.

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`)

Returns The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

```
inline 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.

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`)

Returns The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

```
inline bool start(std::nullptr_t f, int argc, char **argv, init_params const &params = init_params())
```

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

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

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

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`)

Returns The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

```
inline 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.

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`)

Returns The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

hpx/hpx_suspend.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

Note: As long as `ec` is not pre-initialized to `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.

Returns This function will always return zero.

```
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.

Note: As long as `ec` is not pre-initialized to `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.

Returns This function will always return zero.

naming_base

See [Public API](#) for a list of names and headers that are part of the public HPX API.

`hpx/naming_base/unmanaged.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace `hpx`

namespace `naming`

Functions

```
hpx::id_type unmanaged(hpx::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.

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.

Returns 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.

parcelset

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/parcelset/connection_cache.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/parcelset/message_handler_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/parcelset/parcelhandler.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/parcelset/parcelset_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

parcelset_base

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/parcelset_base/parcelport.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/parcelset_base/parcelset_base_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **parcelset**

TypeDefs

```
using parcel_write_handler_type = hpx::function<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:

`enumerator parcelport_background_mode_flush_buffers`

perform buffer flush operations

`enumerator parcelport_background_mode_send`

perform send operations (includes buffer flush)

`enumerator parcelport_background_mode_receive`

perform receive operations

`enumerator parcelport_background_mode_all`

perform all operations

[`hpx/parcelset_base/set_parcel_write_handler.hpp`](#)

See *Public API* for a list of names and headers that are part of the public HPX API.

performance_counters

See *Public API* for a list of names and headers that are part of the public HPX API.

[`hpx/performance_counters/counter_creators.hpp`](#)

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 default_counter_discoverer(counter_info const&, discover_counter_func const&,
                                discover_counters_mode, error_code&)
```

Default discovery function for performance counters; to be registered with the counter types. It will pass the *counter_info* and the *error_code* to the supplied function.

```
bool locality_counter_discoverer(counter_info const&, discover_counter_func const&,
                                discover_counters_mode, error_code&)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/total)/<instancename>

```
bool locality_pool_counter_discoverer(counter_info const&, discover_counter_func const&,
                                         discover_counters_mode, error_code&)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/pool#<pool_name>/total)/<instancename>

```
bool locality0_counter_discoverer(counter_info const&, discover_counter_func const&,
                                   discover_counters_mode, error_code&)
```

Default discoverer function for AGAS performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#0/total}/<instancename>

```
bool locality_thread_counter_discoverer(counter_info const&, discover_counter_func const&,
                                         discover_counters_mode, error_code&)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/worker-thread#<threadnum>)/<instancename>

```
bool locality_pool_thread_counter_discoverer(counter_info const &info,
                                              discover_counter_func const &f,
                                              discover_counters_mode mode, error_code
                                              &ec)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#<locality_id>/pool#<poolname>/thread#<threadnum> }/<instancename>

```
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::function<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::function<std::vector<std::int64_t>(bool)> const&,  
                                              error_code&)
```

```
naming::gid_type agas_raw_counter_creator(counter_info const&, error_code&, char const*const)
```

Creation function for raw counters. The passed function is encapsulating the actual value to monitor. This function checks the validity of the supplied counter name, it has to follow the scheme:

```
/agas(<objectinstance>/total)/<instancename>
```

```
bool agas_counter_discoverer(counter_info const&, discover_counter_func const&,  
                             discover_counters_mode, error_code&)
```

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

```
/agas(<objectinstance>/total)/<instancename>
```

```
naming::gid_type local_action_invocation_counter_creator(counter_info const&,  
                                              error_code&)
```

```
bool local_action_invocation_counter_discoverer(counter_info const&,  
                                              discover_counter_func const&,  
                                              discover_counters_mode, error_code&)
```

hpx/performance_counters/counters.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

```
namespace performance_counters
```

Typedefs

typedef `hpx::function<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::function<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::function<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 class **counter_type**

Values:

enumerator **text**

text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

enumerator **raw**

raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

enumerator **monotonically_increasing**

monotonically_increasing shows the cumulatively accumulated observed value. It does not deliver an average.

Formula: None. Shows cumulatively accumulated data as collected. Average: None Type: Instantaneous

enumerator **average_base**

average_base is used as the base data (denominator) in the computation of time or count averages for the *counter_type::average_count* and *counter_type::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: SUM (N) / x Type: Instantaneous

enumerator **average_count**

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

enumerator **aggregating**

aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: $F(N_x)$

enumerator **average_timer**

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

enumerator **elapsed_time**

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

enumerator **histogram**

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.

enumerator **raw_values**

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.

enumerator **text**

enumerator **raw**

enumerator **monotonically_increasing**

enumerator **average_base**

enumerator **average_count**

enumerator **aggregating**

enumerator **average_timer**

enumerator **elapsed_time**

enumerator **histogram**

enumerator **raw_values**

raw_values counter 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 class **counter_status**

Status and error codes used by the functions related to performance counters.

Values:

enumerator **valid_data**

No error occurred, data is valid.

enumerator **new_data**

Data is valid and different from last call.

enumerator **invalid_data**

Some error occurred, data is not value.

enumerator **already_defined**

The type or instance already has been defined.

enumerator **counter_unknown**

The counter instance is unknown.

enumerator **counter_type_unknown**

The counter type is unknown.

enumerator **generic_error**

A unknown error occurred.

```
enumerator valid_data
enumerator new_data
enumerator invalid_data
enumerator already_defined
enumerator counter_unknown
enumerator counter_type_unknown
enumerator generic_error
```

Functions

```
inline std::string &ensure_counter_prefix(std::string &name)
inline std::string ensure_counter_prefix(std::string const &counter)
inline std::string &remove_counter_prefix(std::string &name)
inline 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.
inline bool status_is_valid(counter_status s)
inline counter_status add_counter_type(counter_info const &info, error_code &ec)
inline hpx::id_type get_counter(std::string const &name, error_code &ec)
inline hpx::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

```
inline counter_info(counter_type type = counter_type::raw)
inline counter_info(std::string const &name)
inline counter_info(counter_type type, std::string const &name, std::string const &helptext = "",  

                    std::uint32_t version = HPX_PERFORMANCE_COUNTER_V1, std::string  

                    const &uom = "")
```

Public Members

counter_type **type_**

The type of the described counter.

std::uint32_t **version_**

The version of the described counter using the 0xMMmmSSSS scheme

counter_status **status_**

The status of the counter object.

std::string **fullname_**

The full name of this counter.

std::string **helptext_**

The full descriptive text for this counter.

std::string **unit_of_measure_**

The unit of measure for this counter.

Private Functions

```
void serialize(serialization::output_archive &ar, const unsigned int)
```

```
void serialize(serialization::input_archive &ar, const unsigned int)
```

Friends

```
friend class 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

```
inline counter_path_elements()
```

```
inline 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)
```

```
inline 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 class hpx::serialization::access
```

```
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

```
inline counter_type_path_elements()
```

```
inline 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 class hpx::serialization::access
```

hpx/performance_counters/counters_fwd.hpp

See [Public API](#) for a list of names and headers that are part of the public *HPX* API.

Defines

```
HPX_COUNTER_TYPE_UNSCOPED_ENUM_DEPRECATED_MSG
```

```
HPX_COUNTER_STATUS_UNSCOPED_ENUM_DEPRECATED_MSG
```

```
HPX_PERFORMANCE_COUNTER_V1
```

```
HPX_DISCOVER_COUNTERS_MODE_UNSCOPED_ENUM_DEPRECATED_MSG
```

namespace **hpx**

```
namespace performance_counters
```

Enums

enum class **counter_type**

Values:

enumerator **text**

text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

enumerator **raw**

raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

enumerator **monotonically_increasing**

monotonically_increasing shows the cumulatively accumulated observed value. It does not deliver an average.

Formula: None. Shows cumulatively accumulated data as collected. Average: None Type: Instantaneous

enumerator `average_base`

average_base is used as the base data (denominator) in the computation of time or count averages for the `counter_type::average_count` and `counter_type::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

enumerator `average_count`

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

enumerator `aggregating`

aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: $F(Nx)$

enumerator `average_timer`

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

enumerator `elapsed_time`

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

enumerator `histogram`

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.

enumerator `raw_values`

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.

enumerator `text`**enumerator `raw`****enumerator `monotonically_increasing`****enumerator `average_base`****enumerator `average_count`****enumerator `aggregating`****enumerator `average_timer`****enumerator `elapsed_time`****enumerator `histogram`****enumerator `raw_values`**

raw_values counter 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 class `counter_status`

Values:

enumerator `valid_data`

No error occurred, data is valid.

enumerator `new_data`

Data is valid and different from last call.

enumerator `invalid_data`

Some error occurred, data is not valid.

enumerator `already_defined`

The type or instance already has been defined.

enumerator **counter_unknown**

The counter instance is unknown.

enumerator **counter_type_unknown**

The counter type is unknown.

enumerator **generic_error**

A unknown error occurred.

enumerator **valid_data**

enumerator **new_data**

enumerator **invalid_data**

enumerator **already_defined**

enumerator **counter_unknown**

enumerator **counter_type_unknown**

enumerator **generic_error**

enum class **discover_counters_mode**

Values:

enumerator **minimal**

enumerator **full**

Functions

inline constexpr bool **operator<(counter_type lhs, counter_type rhs)** noexcept

inline constexpr bool **operator>(counter_type lhs, counter_type rhs)** noexcept

inline *std*::ostream &**operator<<(std::ostream &os, counter_status const &rhs)** noexcept

counter_status **get_counter_type_name(counter_type_path_elements const &path, std::string &result, error_code &ec = throws)**

Create a full name of a counter type from the contents of the given *counter_type_path_elements* instance. The generated counter type name will not contain any parameters.

counter_status **get_full_counter_type_name(counter_type_path_elements const &path, std::string &result, error_code &ec = throws)**

Create a full name of a counter type from the contents of the given *counter_type_path_elements* instance. The generated counter type name will contain all parameters.

```
counter_status get_counter_name(counter_path_elements const &path, std::string &result, error_code&ec = throws)
```

Create a full name of a counter from the contents of the given *counter_path_elements* instance.

```
counter_status get_counter_instance_name(counter_path_elements const &path, std::string &result, error_code&ec = throws)
```

Create a name of a counter instance from the contents of the given *counter_path_elements* instance.

```
counter_status get_counter_type_path_elements(std::string const &name, counter_type_path_elements &path, error_code&ec = throws)
```

Fill the given *counter_type_path_elements* instance from the given full name of a counter type.

```
counter_status get_counter_path_elements(std::string const &name, counter_path_elements &path, error_code&ec = throws)
```

Fill the given *counter_path_elements* instance from the given full name of a counter.

```
counter_status get_counter_name(std::string const &name, std::string &countername, error_code&ec = throws)
```

Return the canonical counter instance name from a given full instance name.

```
counter_status get_counter_type_name(std::string const &name, std::string &type_name, error_code&ec = throws)
```

Return the canonical counter type name from a given (full) instance name.

```
HPX_DEPRECATED_V (1, 9,  
HPX_DISCOVER_COUNTERS_MODE_UNSCOPED_ENUM_DEPRECATION_MSG) inline const expr discover_counters_
```

```
counter_status complement_counter_info(counter_info &info, counter_info const &type_info, error_code&ec = throws)
```

Complement the counter info if parent instance name is missing.

```
counter_status complement_counter_info(counter_info &info, error_code&ec = throws)
```

```
counter_status add_counter_type(counter_info const &info, create_counter_func const &create_counter, discover_counters_func const &discover_counters, error_code&ec = throws)
```

```
counter_status discover_counter_types(discover_counter_func const &discover_counter, discover_counters_mode mode = discover_counters_mode::minimal, error_code&ec = throws)
```

Call the supplied function for each registered counter type.

```
counter_status discover_counter_types(std::vector<counter_info> &counters, discover_counters_mode mode = discover_counters_mode::minimal, error_code&ec = throws)
```

Return a list of all available counter descriptions.

```
counter_status discover_counter_type(std::string const &name, discover_counter_func const &discover_counter, discover_counters_mode mode = discover_counters_mode::minimal, error_code&ec = throws)
```

Call the supplied function for the given registered counter type.

```
counter_status discover_counter_type(counter_info const &info, discover_counter_func const
&discover_counter, discover_counters_mode mode =
discover_counters_mode::minimal, error_code &ec =
throws)
```

```
counter_status discover_counter_type(std::string const &name, std::vector<counter_info>
&counters, discover_counters_mode mode =
discover_counters_mode::minimal, error_code &ec =
throws)
```

Return a list of matching counter descriptions for the given registered counter type.

```
counter_status discover_counter_type(counter_info const &info, std::vector<counter_info>
&counters, discover_counters_mode mode =
discover_counters_mode::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<hpx::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<hpx::id_type> get_counter_async(counter_info const &info, error_code &ec = throws)
```

Get the global id of an existing performance counter, if the counter does not exist yet, the function attempts to create the counter based on the given counter info.

```
void get_counter_infos(counter_info const &info, counter_type &type, std::string &helptext,
std::uint32_t &version, error_code &ec = throws)
```

Retrieve the meta data specific for the given counter instance.

```
void get_counter_infos(std::string name, counter_type &type, std::string &helptext, std::uint32_t
&version, error_code &ec = throws)
```

Retrieve the meta data specific for the given counter instance.

Variables

```
constexpr counter_type counter_text = counter_type::text  
  
constexpr counter_type counter_raw = counter_type::raw  
  
constexpr counter_type counter_monotonically_increasing =  
counter_type::monotonically_increasing  
  
constexpr counter_type counter_average_base = counter_type::average_base  
  
constexpr counter_type counter_average_count = counter_type::average_count  
  
constexpr counter_type counter_aggregating = counter_type::aggregating  
  
constexpr counter_type counter_average_timer = counter_type::average_timer  
  
constexpr counter_type counter_elapsed_time = counter_type::elapsed_time  
  
constexpr counter_type counter_raw_values = counter_type::raw_values  
  
constexpr counter_type counter_histogram = counter_type::histogram  
  
constexpr counter_status status_valid_data = counter_status::valid_data  
  
constexpr counter_status status_new_data = counter_status::new_data  
  
constexpr counter_status status_invalid_data = counter_status::invalid_data  
  
constexpr counter_status status_already_defined = counter_status::already_defined  
  
constexpr counter_status status_counter_unknown = counter_status::counter_unknown  
  
constexpr counter_status status_counter_type_unknown = counter_status::counter_type_unknown  
  
constexpr counter_status status_generic_error = counter_status::generic_error  
  
struct counter_value
```

Public Functions

```
inline counter_value(std::int64_t value = 0, std::int64_t scaling = 1, bool scale_inverse = false)
template<typename T>
inline 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 class hpx::serialization::access
```

```
struct counter_values_array
```

Public Functions

```
inline counter_values_array(std::int64_t scaling = 1, bool scale_inverse = false)

inline counter_values_array(std::vector<std::int64_t> &&values, std::int64_t scaling = 1, bool
                           scale_inverse = false)

inline counter_values_array(std::vector<std::int64_t> const &values, std::int64_t scaling = 1,
                           bool scale_inverse = false)

template<typename T>
inline 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 class hpx::serialization::access
```

hpx/performance_counters/manage_counter_type.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **performance_counters**

Functions

```
counter_status install_counter_type(std::string const &name, hpx::function<std::int64_t(bool)>
                                         const &counter_value, std::string const &helptext = "",
                                         std::string const &uom = "", counter_type type =
                                         counter_type::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*.

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.

Returns If successful, this function returns *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*).

```
counter_status install_counter_type(std::string const &name,
                                    hpx::function<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*.

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.

Returns If successful, this function returns *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*).

```
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.

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.

Returns If successful, this function returns *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*).

```
counter_status install_counter_type(std::string const &name, counter_type type, std::string const  
&helptext, std::string const &uom = "", std::uint32_t version =  
HPX_PERFORMANCE_COUNTER_V1, error_code &ec =  
throws)
```

Install a new performance counter type in a way, which will uninstall it automatically during shutdown.

The function *install_counter_type* will register a new counter type based on the provided *counter_type_info*. The counter type will be automatically unregistered during system shutdown.

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.

Returns If successful, this function returns *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*).

```
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*.

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.

Returns If successful, this function returns *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*).

hpx/performance_counters/registry.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **performance_counters**

 class **registry**

Public Functions

registry() = default

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.

```
inline 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::function<std::int64_t()>
                                   const &f, naming::gid_type &id, error_code &ec = throws)
```

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

```
counter_status create_raw_counter(counter_info const &info, hpx::function<std::int64_t(bool)>
                                   const &f, naming::gid_type &id, error_code &ec = throws)
```

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

```
counter_status create_raw_counter(counter_info const &info,
                                   hpx::function<std::vector<std::int64_t>()> const &f,
                                   naming::gid_type &id, error_code &ec = throws)
```

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

```
counter_status create_raw_counter(counter_info const &info,
                                   hpx::function<std::vector<std::int64_t>(bool)> const &f,
                                   naming::gid_type &id, error_code &ec = throws)
```

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

```
counter_status create_counter(counter_info const &info, naming::gid_type &id, error_code &ec = throws)
```

Create a new performance counter instance based on given counter info.

```
counter_status create_statistics_counter(counter_info const &info, std::string const &base_counter_name, std::vector<std::size_t> const &parameters, naming::gid_type &id, error_code &ec = throws)
```

Create a new statistics performance counter instance based on given base counter name and given base time interval (milliseconds).

```
counter_status create_arithmetics_counter(counter_info const &info, std::vector<std::string> const &base_counter_names, naming::gid_type &id, error_code &ec = throws)
```

Create a new arithmetics performance counter instance based on given base counter names.

```
counter_status create_arithmetics_counter_extended(counter_info const &info, std::vector<std::string> const &base_counter_names, naming::gid_type &id, error_code &ec = throws)
```

Create a new extended arithmetics performance counter instance based on given base counter names.

```
counter_status add_counter(hpx::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, hpx::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

```
inline 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_
```

plugin_factories

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/plugin_factories/binary_filter_factory.hpp

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 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

```
inline 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() override = default
```

```
inline serialization::binary_filter *create(bool compress, serialization::binary_filter *next_filter = nullptr) override
```

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_
```

hpx/plugin_factories/message_handler_factory.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/plugin_factories/parcelport_factory.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/plugin_factories/plugin_registry.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

Defines

HPX_REGISTER_PLUGIN_REGISTRY(...)

This macro is used create and to 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

inline bool **get_plugin_info**(*std*::vector<*std*::string> &fillini) override

Return the ini-information for all contained components.

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.

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

runtime_components

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/runtime_components/component_factory.hpp

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 create and to register a minimal component factory for a component type which allows it to be remotely created using the `hpx::new_<>` function.

This macro can be invoked with one, two or three arguments

Parameters

- **type** – The *type* parameter is a (fully decorated) type of the component type for which a factory should be defined.
- **name** – The *name* parameter specifies the name to use to register the factory. This should uniquely (system-wide) identify the component type. The *name* parameter must conform to the C++ identifier rules (without any namespace). If this parameter is not given, the first parameter is used.
- **mode** – The *mode* parameter has to be one of the defined enumeration values of the enumeration `hpx::components::factory_state_enum`. The default for this parameter is `hpx::components::factory_enabled`.

hpx/runtime_components/component_registry.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

Defines

`HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY`(...)

This macro is used create and 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

```
inline bool get_component_info(std::vector<std::string> &fillini, std::string const &filepath, bool
is_static = false) override
```

Return the ini-information for all contained components.

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.

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

```
inline void register_component_type() override
```

Return the unique identifier of the component type this factory is responsible for.

Parameters

- **locality** – [in] The id of the locality this factory is responsible for.
- **agas_client** – [in] The AGAS client to use for component id registration (if needed).

Returns Returns the unique identifier of the component type this factory instance is responsible for. This function throws on any error.

hpx/runtime_components/components_fwd.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
components::server::runtime_support *get_runtime_support_ptr()
```

namespace **components**

```
template<typename Component>
struct component_factory
```

namespace **server**

namespace **stubs**

namespace **components**

hpx/runtime_components/default_distribution_policy.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

Variables

static 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

Default-construct a new instance of a *default_distribution_policy*. This policy will represent one locality (the local locality).

inline *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

inline *default_distribution_policy* **operator()** (*std*::vector<*id_type*> &&loc) const

Create a new *default_distribution* policy representing the given set of localities.

Parameters **loc** – [in] The locality the new instance should represent

inline *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*>

inline *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

Parameters `vs` – [in] The arguments which will be forwarded to the constructor of the new object.

Returns A future holding the global address which represents the newly created object

```
template<typename Component, typename ...Ts>
inline 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

Parameters

- `count` – [in] The number of objects to create
- `vs` – [in] The arguments which will be forwarded to the constructors of the new objects.

Returns A future holding the list of global addresses which represent the newly created objects

```
template<typename Action, typename ...Ts>
inline async_result<Action>::type async(launch policy, Ts&&... vs) const
```

```
template<typename Action, typename Callback, typename ...Ts>
inline 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>
inline 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>
inline bool apply(threads::thread_priority priority, Ts&&... vs) const
```

```
template<typename Action, typename Continuation, typename Callback, typename ...Ts>
inline 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>
inline bool apply_cb(threads::thread_priority priority, Callback &&cb, Ts&&... vs) const
```

```
inline 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

```
inline hpx::id_type get_next_target() const
```

Returns the locality which is anticipated to be used for the next async operation

```
template<typename Action>
struct async_result
#include <default_distribution_policy.hpp>
```

Note: This function is part of the invocation policy implemented by this class

Public Types

```
using type = hpx::future<typename traits::promise_local_result<typename
hpx::traits::extract_action<Action>::remote_result_type>::type>
```

hpx/runtime_components/derived_component_factory.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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)
```

hpx/runtime_components/new.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

```
template<typename Component, typename... Ts> < unspecified > new_ (id_type const &locality, Ts &&... vs)
```

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

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

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

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

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.

Returns 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.

```
template<typename Component, typename... Ts> < unspecified > 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();
```

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.

Returns 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.

```
template<typename Component, typename... Ts> <unspecified> new_ (id_type const &locality, std::size_t count, Ts &&... vs)
```

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

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

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

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

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.

Returns 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.

```
template<typename Component, typename DistPolicy, typename...
Ts> < unspecified > new_ (DistPolicy const &policy, Ts &&... vs)
```

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

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

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

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

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.

Returns 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.

```
template<typename Component, typename DistPolicy, typename...
Ts> < unspecified > new_ (DistPolicy const &policy, std::size_t count, Ts &&... vs)
```

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

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

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

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

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.

Returns 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.

runtime_distributed

See [Public API](#) for a list of names and headers that are part of the public HPX API.

hpx/runtime_distributed.hpp

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

```
explicit runtime_distributed(util::runtime_configuration &rtcfg, int (*pre_main)(runtime_mode) = nullptr, void (*post_main)() = nullptr)
```

Construct a new HPX runtime instance

Parameters **locality_mode** – [in] This is the mode the given runtime instance should be executed in.

```
~runtime_distributed()
```

The destructor makes sure all HPX runtime services are properly shut down before exiting.

```
int start(hpx::function<hpx_main_function_type> const &func, bool blocking = false) override
```

Start the runtime system.

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.

Returns 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.

`int start(bool blocking = false) override`

Start the runtime system.

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 .

Returns 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.

`int wait() override`

Wait for the shutdown action to be executed.

Returns 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) override`

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) override`

`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() override`

Suspend the runtime system.

`int resume() override`

Resume the runtime system.

`bool report_error(std::size_t num_thread, std::exception_ptr const &e, bool terminate_all = true) override`

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) override
    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(hpx::function<hpx_main_function_type> const &func) override
```

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.

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.

Returns This function will return the value as returned as the result of the invocation of the function object given by the parameter **func**.

```
int run() override
```

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.

Returns This function will always return 0 (zero).

```
bool is_networking_enabled() override
```

```
template<typename F>
inline 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() override
    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 override
    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) override
    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) override
    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) override
    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) override
    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) override
    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) override
    Register an external OS-thread with HPX.
notification_policy_type get_notification_policy(char const *prefix,
    runtime_local::os_thread_type type) override
    Generate a new notification policy instance for the given thread name prefix
std::uint32_t get_locality_id(error_code &ec) const override
std::size_t get_num_worker_threads() const override
std::uint32_t get_num_localities(hpx::launch::sync_policy, error_code &ec) const override
std::uint32_t get_initial_num_localities() const override
hpx::future<std::uint32_t> get_num_localities() const override
std::string get_locality_name() const override
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) override
std::uint32_t assign_cores() override
```

Private Types

```
using used_cores_map_type = std::map<std::string, std::uint32_t>
```

Private Functions

```
threads::thread_result_type run_helper(hpx::function<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&)
```

hpx/runtime_distributed/applier.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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() = default

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.

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.

Returns The function returns *true* if there is at least one remote locality known to the AGASservice (!prefixes.empty()).

**bool get_remote_localities(*std::vector<hpX::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.

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.

Returns The function returns *true* if there is at least one remote locality known to the AGASservice (!prefixes.empty()).

```
bool get_localities(std::vector<hpx::id_type> &locality_ids, error_code &ec = throws) const  
bool get_localities(std::vector<hpx::id_type> &locality_ids, components::component_type  
type, error_code &ec = throws) const  
inline 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 run-  
time_support is responsible for  
inline hpx::id_type const &get_runtime_support_gid() const  
By convention the runtime_support has a gid identical to the prefix of the locality the run-  
time_support is responsible for
```

Private Members

threads::*threadmanager* ***thread_manager_**

hpx::id_type **runtime_support_id_**

hpx/runtime_distributed/applier_fwd.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

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.

hpx/runtime_distributed/copy_component.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

Functions

```
template<typename Component>
future<hpx::id_type> copy(hpx::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.

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.

Returns A future representing the global id of the newly (copied) component instance.

```
template<typename Component>
future<hpx::id_type> copy(hpx::id_type const &to_copy, hpx::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.

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.

Returns A future representing the global id of the newly (copied) component instance.

```
template<typename Derived, typename Stub>
Derived copy(client_base<Derived, Stub> const &to_copy, hpx::id_type const &target_locality =
hpx::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.

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.

Returns A future representing the global id of the newly (copied) component instance.

hpx/runtime_distributed/find_all_localities.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

hpx::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.

See also:

[hpx::find_all_localities\(\)](#), [hpx::find_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).

Note: As long as *ec* is not pre-initialized to *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::invalid_id* otherwise.

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.

Returns The global id representing the root locality for this application.

std::vector<hpx::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.

See also:

[hpx::find_here\(\)](#), [hpx::find_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).

Note: As long as *ec* is not pre-initialized to *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.

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.

Returns The global ids representing the localities currently available to this application.

std::vector<hpx::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).

See also:

hpx::find_here(), *hpx::find_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).

Note: As long as *ec* is not pre-initialized to *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.

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.

Returns The global ids representing the remote localities currently available to this application.

hpx/runtime_distributed/find_here.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

`hpx::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.

See also:

`hpx::find_all_localities()`, `hpx::find_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).

Note: As long as `ec` is not pre-initialized to `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::invalid_id` otherwise.

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.

Returns The global id representing the locality this function has been called on.

hpx/runtime_distributed/find_localities.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

`std::vector<hpx::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.

See also:

`hpx::find_here()`, `hpx::find_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).

Note: As long as `ec` is not pre-initialized to `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.

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.

Returns 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.

`std::vector<hpx::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.

See also:

`hpx::find_here()`, `hpx::find_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).

Note: As long as *ec* is not pre-initialized to *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.

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.

Returns The global ids representing the remote localities currently available to this application.

*hpx::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.

See also:

hpx::find_here(), hpx::find_all_localities()

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).

Note: As long as *ec* is not pre-initialized to *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::invalid_id* otherwise.

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.

Returns 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::invalid_id`.

hpx/runtime_distributed/get_locality_name.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

Functions

future<std::string> **get_locality_name**(*hpx::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.

See also:

std::string get_locality_name()

Parameters **id** – [in] The global id of the locality for which the name should be retrieved

Returns 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.

hpx/runtime_distributed/get_num_localities.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

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.

See also:

hpx::find_all_localities, hpx::get_num_localities

Note: This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

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.

See also:

hpx::find_all_localities, hpx::get_num_localities

Note: This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

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.

hpx/runtime_distributed/migrate_component.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

namespace **components**

Functions

```
template<typename Component, typename DistPolicy>
future<hpx::id_type> migrate(hpx::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.

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.

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

```
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.

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.

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

```
template<typename Component>
future<hpx::id_type> migrate(hpx::id_type const &to_migrate, hpx::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.

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.

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

```
template<typename Derived, typename Stub>
inline Derived migrate(client_base<Derived, Stub> const &to_migrate, hpx::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.

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.

Returns A client side representation of representing of the migrated component instance. This should be the same as *migrate_to*.

hpx/runtime_distributed/runtime_fwd.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

hpx/runtime_distributed/runtime_support.hpp

See *Public API* for a list of names and headers that are part of the public *HPX API*.

namespace **hpx**

namespace **agas**

Functions

```
struct runtime_components_init_interface_functions &runtime_components_init()
```

namespace **components**

Functions

```
struct counter_interface_functions &counter_init()
```

```
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

```
inline runtime_support(hpx::id_type const &gid = hpx::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>
```

```
inline hpx::id_type create_component(Ts&&... vs)
```

Create a new component type using the `runtime_support`.

```
template<typename Component, typename ...Ts>
```

```
inline hpx::future<hpx::id_type> create_component_async(Ts&&... vs)
```

Asynchronously create a new component using the `runtime_support`.

```
template<typename Component, typename ...Ts>
```

```
inline std::vector<hpx::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>
```

```
inline hpx::future<std::vector<hpx::id_type>> bulk_create_components_async(std::size_t, Ts&&... vs)
```

Asynchronously create a new component using the `runtime_support`.

```
inline hpx::future<int> load_components_async()
inline int load_components()

inline hpx::future<void> call_startup_functions_async(bool pre_startup)
inline void call_startup_functions(bool pre_startup)

inline hpx::future<void> shutdown_async(double timeout = -1)
    Shutdown the given runtime system.
inline void shutdown(double timeout = -1)

inline void shutdown_all(double timeout = -1)
    Shutdown the runtime systems of all localities.

inline hpx::future<void> terminate_async()
    Terminate the given runtime system.
inline void terminate()

inline void terminate_all()
    Terminate the runtime systems of all localities.

inline void get_config(util::section &ini)
    Retrieve configuration information.

inline hpx::id_type const &get_id() const
inline naming::gid_type const &get_raw_gid() const
```

Private Types

```
typedef stubs::runtime_support base_type
```

Private Members

```
hpx::id_type gid_
```

hpx/runtime_distributed/server/copy_component.hpp

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

namespace **server**

Functions

```
template<typename Component>
future<hpx::id_type> copy_component_here(hpx::id_type const &to_copy)

template<typename Component>
future<hpx::id_type> copy_component(hpx::id_type const &to_copy, hpx::id_type const
&target_locality)

template<typename Component>
struct copy_component_action : public hpx::actions::action<future<hpx::id_type>>
(*)(hpx::id_type const&, hpx::id_type const&), &copy_component<Component>,
copy_component_action<Component>>

template<typename Component>
struct copy_component_action_here : public hpx::actions::action<future<hpx::id_type>>
(*)(hpx::id_type const&), &copy_component_here<Component>,
copy_component_action_here<Component>>
```

hpx/runtime_distributed/server/runtime_support.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **components**

 namespace **server**

 class **runtime_support**

Public Types

typedef *runtime_support* **type_holder**

Public Functions

```
explicit runtime_support(hpx::util::runtime_configuration &cfg)
inline ~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)  
  
template<typename Component>  
naming::gid_type migrate_component_to_here(std::shared_ptr<Component> const &p,  
                                              hpx::id_type)  
  
void shutdown(double timeout, hpx::id_type const &respond_to)  
    Gracefully shutdown this runtime system instance.  
  
void shutdown_all(double timeout)  
    Gracefully shutdown runtime system instances on all localities.  
  
void terminate(hpx::id_type const &respond_to)  
    Shutdown this runtime system instance.  
  
inline void terminate_act(hpx::id_type const &id)  
  
void terminate_all()  
    Shutdown runtime system instances on all localities.  
  
inline 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) HPX_DEFINE_COMPONENT_ACTION(runtime_support
termination detection

terminate_all_action HPX_DEFINE_COMPONENT_ACTION (runtime_support,
remove_from_connection_cache) void run()
Start the runtime_support component.

void wait()
Wait for the runtime_support component to notify the calling thread.

This function will be called from the main thread, causing it to block while the HPX functionality is executed. The main thread will block until the shutdown_action is executed, which in turn notifies all waiting threads.

void stop(double timeout, hpx::id_type const &respond_to, bool remove_from_remote_caches)
Notify all waiting (blocking) threads allowing the system to be properly stopped.
```

Note: This function can be called from any thread.

```
void stopped()
called locally only

void notify_waiting_main()

inline 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

terminate_all_act

Public Static Functions

```
static inline component_type get_component_type()

static inline void set_component_type(component_type t)
```

```
static inline constexpr void finalize()  
    finalize() will be called just before the instance gets destructed  
    Parameters  
        • self – [in] The HPX thread used to execute this function.  
        • app1 – [in] The applier to be used for finalization of the component instance.  
  
static inline bool is_target_valid(hpx::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<hpx::id_type> const
                                            &locality_ids)
```

Private Types

```
typedef hpx::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_
hpx::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

```
inline 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
```

hpx/runtime_distributed/stubs/runtime_support.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **components**

 namespace **stubs**

 struct **runtime_support**

 Subclassed by *hpx::components::runtime_support*

Public Static Functions

```
template<typename Component, typename ...Ts>
static inline hpx::future<hpx::id_type> create_component_async(hpx::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 inline hpx::id_type create_component(hpx::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 inline hpx::future<std::vector<hpx::id_type>> bulk_create_component_colocated_async(hpx::id_type
const &gid,
std::size_t count,
Ts&&...
vs)
```

Create multiple new components *type* using the *runtime_support* colocated with the with the given *targetgid*. This is a non-blocking call.

```
template<typename Component, typename ...Ts>
static inline std::vector<hpx::id_type> bulk_create_component_colocated(hpx::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 inline hpx::future<std::vector<hpx::id_type>> bulk_create_component_async(hpx::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 inline std::vector<hpx::id_type> bulk_create_component(hpx::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 inline hpx::future<hpx::id_type> create_component_colocated_async(hpx::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 inline hpx::id_type create_component_colocated(hpx::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 inline hpx::future<hpx::id_type> copy_create_component_async(hpx::id_type const &gid,
                                                               std::shared_ptr<Component> const &p, bool local_op)
```

```
template<typename Component>
static inline hpx::id_type copy_create_component(hpx::id_type const &gid,
                                                 std::shared_ptr<Component> const &p,
                                                 bool local_op)
```

```
template<typename Component>
static inline hpx::future<hpx::id_type> migrate_component_async(hpx::id_type const &target_locality,
                                                               std::shared_ptr<Component> const &p, hpx::id_type const &to_migrate)
```

```
template<typename Component, typename DistPolicy>
static inline hpx::future<hpx::id_type> migrate_component_async(DistPolicy const &policy,
                                                               std::shared_ptr<Component> const &p, hpx::id_type const &to_migrate)
```

```
template<typename Component, typename Target>
static inline hpx::id_type migrate_component(Target const &target, hpx::id_type const &to_migrate, std::shared_ptr<Component> const &p)
```

```
static hpx::future<int> load_components_async(hpx::id_type const &gid)
```

```
static int load_components(hpx::id_type const &gid)
```

```
static hpx::future<void> call_startup_functions_async(hpx::id_type const &gid, bool pre_startup)
```

```
static void call_startup_functions(hpx::id_type const &gid, bool pre_startup)
```

```
static hpx::future<void> shutdown_async(hpx::id_type const &targetgid, double timeout = -1)
    Shutdown the given runtime system.
```

```
static void shutdown(hpx::id_type const &targetgid, double timeout = -1)
```

```
static void shutdown_all(hpx::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(hpx::id_type const &targetgid)
    Retrieve configuration information.

    Terminate the given runtime system

static void terminate(hpx::id_type const &targetgid)

static void terminate_all(hpx::id_type const &targetgid)
    Terminate the runtime systems of all localities.

static void terminate_all()

static void garbage_collect_non_blocking(hpx::id_type const &targetgid)

static hpx::future<void> garbage_collect_async(hpx::id_type const &targetgid)

static void garbage_collect(hpx::id_type const &targetgid)

static hpx::future<hpx::id_type> create_performance_counter_async(hpx::id_type
targetgid, performance_counters::counter_info
const &info)
```

const &info, *error_code &ec = throws*)

```

static hpx::id_type create_performance_counter(hpx::id_type targetgid,
performance_counters::counter_info
const &info, error_code &ec = throws)
```

```

static hpx::future<util::section> get_config_async(hpx::id_type const &targetgid)
    Retrieve configuration information.

static void get_config(hpx::id_type const &targetgid, util::section &ini)

static void remove_from_connection_cache_async(hpx::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.

hpx/parallel/segmented_algorithms/adjacent_difference.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 namespace **segmented**

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> tag_invoke(hpx::adjacent_difference_t,
    ExPolicy &&policy,
    FwdIter1 first,
    FwdIter1 last, FwdIter2
    dest, Op &&op)

template<typename InIter1, typename InIter2, typename Op>
InIter2 tag_invoke(hpx::adjacent_difference_t, InIter1 first, InIter1 last, InIter2 dest, Op &&op)
```

`hpx/parallel/segmented_algorithms/adjacent_find.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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)
```

`hpx/parallel/segmented_algorithms/all_any_none.hpp`

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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)
```

hpx/parallel/segmented_algorithms/count.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 namespace **segmented**

Functions

```
template<typename InIter, typename T>
std::iterator_traits<InIter>::difference_type tag_invoke(hpx::count_t, InIter first, InIter last, T const
    &value)

template<typename ExPolicy, typename SegIter, typename T>
```

hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<SegIter>::difference_type>::type tag_invoke

```
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_invoke
```

[hpx/parallel/segmented_algorithms/exclusive_scan.hpp](#)

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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())
```

hpx/parallel/segmented_algorithms/fill.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

hpx/parallel/segmented_algorithms/for_each.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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)
```

hpx/parallel/segmented_algorithms/generate.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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)
```

hpx/parallel/segmented_algorithms/inclusive_scan.hpp

See *Public API* for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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>>
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)
```

hpx/parallel/segmented_algorithms/minmax.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

Typedefs

```
template<typename T>
using minmax_element_result = hpx::parallel::util::min_max_result<T>
```

 namespace **v1**

 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_ele-
    icy
    &&pol-
    icy,
    Se-
    gIter
    first,
    Se-
    gIter
    last,
    F
    &&f)
```

hpx/parallel/segmented_algorithms/reduce.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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)
```

hpx/parallel/segmented_algorithms/transform.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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
```

```
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

```
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
```

hpx/parallel/segmented_algorithms/transform_exclusive_scan.hpp

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 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 &&policy,
FwdIter1 first, FwdIter1
last, FwdIter2 dest, T
init, Op &&op, Conv
&&conv)
```

hpx/parallel/segmented_algorithms/transform_inclusive_scan.hpp

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 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)
```

hpx/parallel/segmented_algorithms/transform_reduce.hpp

See [Public API](#) for a list of names and headers that are part of the public HPX API.

namespace **hpx**

 namespace **parallel**

 namespace **v1**

 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_reduce_
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 **Convert**>

```

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)

```

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 document⁵²⁰. This is a living document that is constantly updated with relevant information.

⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/blob/master/.github/CONTRIBUTING.md>

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 document⁵²¹ 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
 - Update grant information
5. This step does not apply to patch releases. For both APEX and libCDS:
 - Change the release branch to be the most current release tag available in the APEX/libCDS `git_external` section in the main `CMakeLists.txt`. Please contact the maintainers of the respective packages to generate a new release to synchronize with the *HPX* release ([APEX](#)⁵²², [libCDS](#)⁵²³).
6. Make sure `HPX_VERSION_MAJOR/MINOR/SUBMINOR` in `CMakeLists.txt` contain the correct values. Change them if needed.
7. Change version references in `CITATION.cff`. There are two occurrences.
8. 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:

⁵²¹ <http://hpx.stellar-group.org/documents/governance/>

⁵²² <http://github.com/UO-OACISS/xpress-apex>

⁵²³ <https://github.com/STELLAR-GROUP/libcds>

- 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).

9. Update the minimum required versions if necessary (compilers, dependencies, etc.) in `building_hpx.rst`.
10. Verify that the Jenkins setups for the release branch on Rostam and Piz Daint are running and do not display any errors.
11. Repeat the following steps until satisfied with the release.
 1. Change HPX_VERSION_TAG in `CMakeLists.txt` to `-rcN`, where N is the current iteration of this step. Start with `-rc1`.
 2. Create a pre-release on GitHub using the script `tools/roll_release.sh`. This script automatically tag with the corresponding release number. The script requires that you have the STE||AR Group signing key.
 3. This step is not necessary for patch releases. Notify `hpx-users@stellar-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.
12. 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.
13. 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.
14. Change HPX_VERSION_TAG in `CMakeLists.txt` to an empty string.
15. 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`.
16. Create a release on GitHub using the script `tools/roll_release.sh`. This script automatically tag the with the corresponding release number. The script requires that you have the STE||AR Group signing key.
17. Update the websites ([⁵²⁴](https://hpx.stellar-group.org), [⁵²⁵](https://stellar-group.org) and [⁵²⁶](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).

⁵²⁴ <https://hpx.stellar-group.org>

⁵²⁵ <https://stellar-group.org>

⁵²⁶ <https://stellar.cct.lsu.edu>

18. Merge release branch into master.
19. 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.
20. 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
21. 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
22. 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, our list of external collaborators, isocpp.org, reddit.com, HPC Wire, Inside HPC, Heise Online, and a CCT press release.
23. 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 CTest⁵²⁷ tool and are executed automatically on each commit to the *HPX* [Github⁵²⁸](https://github.com/STELLAR-GROUP/hpx) 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* Issues⁵²⁹ tracker with detailed information about the target system.

Running tests manually

Running the tests manually is as easy as typing `make tests && make test`. This will build all tests and run them once the tests are built successfully. After the tests have been built, you can invoke separate tests with the help of the `ctest` command. You can list all available test targets using `make help | grep tests`. Please see the CTest Documentation⁵³⁰ 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`.

⁵²⁷ <https://gitlab.kitware.com/cmake/community/wikis/doc/ctest/Testing-With-CTest>

⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/>

⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/issues>

⁵³⁰ <https://www.cmake.org/cmake/help/latest/manual/ctest.1.html>

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 Issues⁵³¹](#) page. For more information on how to submit support requests or other means of getting in contact with the developers, please see the [Support Website⁵³²](#) 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 [CircleCI⁵³³](#) and a combination of [CDash⁵³⁴](#) and [pycicle⁵³⁵](#). You can see the dashboards here: [CircleCI HPX dashboard⁵³⁶](#) and [CDash HPX dashboard⁵³⁷](#).

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, [Docker⁵³⁸](#) provides a convenient alternative to quickly get all the dependencies needed to start development of *HPX*. Our testing setup on [CircleCI⁵³⁹](#) uses a docker image to run all tests.

To get started you need to install [Docker⁵⁴⁰](#) using whatever means is most convenient on your system. Once you have [Docker⁵⁴¹](#) installed, you can pull or directly run the docker image. The image is based on Debian and Clang, and can be found on [Docker Hub⁵⁴²](#). 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 Documentation⁵⁴³](#) for more information on using [Docker⁵⁴⁴](#).

⁵³¹ <https://github.com/STELLAR-GROUP/hpx/issues>

⁵³² <https://stellar.cct.lsu.edu/support/>

⁵³³ <https://circleci.com>

⁵³⁴ <https://www.kitware.com/cdash/project/about.html>

⁵³⁵ <https://github.com/biddisco/pycicle/>

⁵³⁶ <https://circleci.com/gh/STELLAR-GROUP/hpx>

⁵³⁷ <https://cdash.cscs.ch/index.php?project=HPX>

⁵³⁸ <https://www.docker.com>

⁵³⁹ <https://circleci.com>

⁵⁴⁰ <https://www.docker.com>

⁵⁴¹ <https://www.docker.com>

⁵⁴² https://hub.docker.com/r/stellargroup/build_env/

⁵⁴³ <https://docs.docker.com/>

⁵⁴⁴ <https://www.docker.com>

Warning: All changes made within the container are lost when the container is closed. If you want files to persist (e.g., the *HPX* source tree) after closing the container, you can bind directories from the host system into the container (see [Docker Documentation \(Bind mounts\)](#)⁵⁴⁵).

2.9.6 Documentation

This documentation is built using [Sphinx](#)⁵⁴⁶, and an automatically generated API reference using [Doxygen](#)⁵⁴⁷ and [Breathe](#)⁵⁴⁸.

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 4.5.0` (Python package)
- `sphinx-book-theme` (Python package)
- `breathe 4.33.1` (Python package)
- `doxygen`
- `sphinxcontrib-bibtex`

If the [Python](#)⁵⁴⁹ dependencies are not available through your system package manager, you can install them using the Python package manager pip:

```
pip install --user "sphinx<5" sphinx-book-theme breathe sphinxcontrib-bibtex
```

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

`DOXYGEN_ROOT:PATH`

Specifies where to look for the installation of the [Doxygen](#)⁵⁵⁰ tool.

`SPHINX_ROOT:PATH`

Specifies where to look for the installation of the [Sphinx](#)⁵⁵¹ tool.

`BREATHE_APIDOC_ROOT:PATH`

Specifies where to look for the installation of the [Breathe](#)⁵⁵² tool.

⁵⁴⁵ <https://docs.docker.com/storage/bind-mounts/>

⁵⁴⁶ <http://www.sphinx-doc.org>

⁵⁴⁷ <https://www.doxygen.org>

⁵⁴⁸ <https://breathe.readthedocs.io/en/latest>

⁵⁴⁹ <https://www.python.org>

⁵⁵⁰ <https://www.doxygen.org>

⁵⁵¹ <http://www.sphinx-doc.org>

⁵⁵² <https://breathe.readthedocs.io/en/latest>

Building documentation

Enable building of the documentation by setting `HPX_WITH_DOCUMENTATION=ON` during CMake⁵⁵³ 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.

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`⁵⁵⁴ 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

⁵⁵³ <https://www.cmake.org>

⁵⁵⁴ https://github.com/STELLAR-GROUP/hpx/blob/master/libs/create_library_skeleton.py

```
– 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
```

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

A main `CMakeLists.txt` is created in the root directory of the module. By default it contains a call to `add_hpx_module` which takes care of most of the boilerplate required for a module. You only need to fill in the source and header files in most cases.

`add_hpx_module` requires a module name. Optional flags are:

Optional single-value arguments are:

- `INSTALL_BINARIES`: Install the resulting library.

Optional multi-value arguments are:

- `SOURCES`: List of source files.
- `HEADERS`: List of header files.
- `COMPAT_HEADERS`: List of compatibility header files.
- `DEPENDENCIES`: Libraries that this module depends on, such as other modules.
- `CMAKE_SUBDIRS`: List of subdirectories to add to the module.

The `include` directory should contain only headers that other libraries need. For each of those headers, an automatic header test to check for self containment will be generated. Private headers should be placed under the `src` directory. This allows for clear separation. The `cmake` subdirectory may include additional `CMake`⁵⁵⁵ scripts needed to generate the respective build configurations.

⁵⁵⁵ <https://www.cmake.org>

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.

Tests are placed in suitable subdirectories of `tests`.

When in doubt, consult existing modules for examples on how to structure the module.

Finding circular dependencies

Our CI will perform a check to see if there are circular dependencies between modules. In cases where it's not clear what is causing the circular dependency, running the `cpp-dependencies`⁵⁵⁶ 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:

```
$ 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
```

⁵⁵⁶ <https://github.com/tomtom-international/cpp-dependencies>

2.10 Releases

2.10.1 HPX V1.9.0 (TBD)

General changes

Breaking changes

- Stopped supporting Clang V8, the minimal version supported is now Clang V10
- Stopped supporting Visual Studio 2015, the minimal version supported is now Visual Studio 2019

Closed issues

Closed pull requests

2.10.2 HPX V1.8.1 (Aug 5, 2022)

This is a bugfix release with a few minor additions and resolved problems.

General changes

This patch release adds a number of small new features and fixes a handful of problems discovered since the last release, in particular:

- A lot of work has been done to improve vectorization support for our parallel algorithms. HPX now supports using EVE - the Expressive Vector Engine as a vectorization backend.
- Added a simple average power consumption performance counter.
- Added performance counters related to the use of zero-copy chunks in the networking layer.
- More work was done towards full compatibility with the sender/receivers proposal P2300.
- Fixing sync_wait to decay the result types
- Fixed collective operations to properly avoid overlapping consecutive operations on the same communicator.
- Simplified the implementation of our execution policies and added mapping functions between those.
- Fixed performance issues with our implementation of *small_vector*.
- Serialization now works with buffers of unsigned characters.
- Fixing dangling reference in serialization of non-default constructible types
- Fixed static linking on Windows.
- Fixed support for M1/MacOS based architectures.
- Fixed support for gentoo/musl.
- Fixed *hpx::counting_semaphore_var*.
- Properly check start and end bounds for *hpx::for_loop*
- A lot of changes and fixes to the documentation (see <https://hpx-docs.stellar-group.org>).

Breaking changes

- No breaking changes have been introduced.

Closed issues

- Issue #5964⁵⁵⁷ - component with multiple inheritance
- Issue #5946⁵⁵⁸ - dll_dlopen.hpp: error: RTLD_DI_ORIGIN was not declared in this scope with musl libc
- Issue #5925⁵⁵⁹ - Simplify implementation of execution policies
- Issue #5924⁵⁶⁰ - {what}: mmap() failed to allocate thread stack: HPX(unhandled_exception)
- Issue #5912⁵⁶¹ - collectives all gather hangs if rank 0 is not involved
- Issue #5902⁵⁶² - MPI parcelport issue on Fugaku
- Issue #5900⁵⁶³ - Unable to build hello_world_distributed.cpp.
- Issue #5892⁵⁶⁴ - Problems with HPX serialization as a standalone feature. Testcase provided.
- Issue #5886⁵⁶⁵ - Segfault when serializing non default constructible class with stl containers data members
- Issue #5832⁵⁶⁶ - Distributed execution crash
- Issue #5768⁵⁶⁷ - HPX hangs on Perlmutter
- Issue #5735⁵⁶⁸ - hpx::for_loop executes without checking start and end bounds
- Issue #5700⁵⁶⁹ - HPX(serialization_error)

Closed pull requests

- PR #5970⁵⁷⁰ - Fixing component multiple inheritance
- PR #5969⁵⁷¹ - Fixing sync_wait to avoid dangling references
- PR #5963⁵⁷² - Fixing sync_wait to decay the result types
- PR #5960⁵⁷³ - docs: added name to documentation contributors list
- PR #5959⁵⁷⁴ - Fixing sync_wait to decay the result types
- PR #5954⁵⁷⁵ - refactor: rename itr to correct type (*reduce*)

⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5964>

⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5946>

⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5925>

⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5924>

⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/5912>

⁵⁶² <https://github.com/STELLAR-GROUP/hpx/issues/5902>

⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/5900>

⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5892>

565 <https://github.com/STELLAR-GROUP/hpx/issues/5886>566 <https://github.com/STELLAR-GROUP/hpx/issues/5832>567 <https://github.com/STELLAR-GROUP/hpx/issues/5768>568 <https://github.com/STELLAR-GROUP/hpx/issues/5735>569 <https://github.com/STELLAR-GROUP/hpx/issues/5700>570 <https://github.com/STELLAR-GROUP/hpx/pull/5970>571 <https://github.com/STELLAR-GROUP/hpx/pull/5969>572 <https://github.com/STELLAR-GROUP/hpx/pull/5963>573 <https://github.com/STELLAR-GROUP/hpx/pull/5960>574 <https://github.com/STELLAR-GROUP/hpx/pull/5959>575 <https://github.com/STELLAR-GROUP/hpx/pull/5954>

- PR #5954⁵⁷⁶ - refactor: rename itr to correct type (*reduce*)
- PR #5953⁵⁷⁷ - Fixed property handling in hierarchical_spawning
- PR #5951⁵⁷⁸ - Fixing static linking (for Windows)
- PR #5947⁵⁷⁹ - Fix building on musl.
- PR #5944⁵⁸⁰ - added adaptive_static_chunk_size
- PR #5943⁵⁸¹ - Fix sync_wait
- PR #5942⁵⁸² - Fix doc warnings
- PR #5941⁵⁸³ - Fix sync_wait
- PR #5940⁵⁸⁴ - Protect collective operations against std::vector<bool> idiosyncrasies
- PR #5939⁵⁸⁵ - docs: fix & improve parallel algorithms documentation 2
- PR #5938⁵⁸⁶ - Properly implement generation support for collective operations
- PR #5937⁵⁸⁷ - Remove leftover files from PMR based small_vector
- PR #5936⁵⁸⁸ - Adding mapping functions between execution policies
- PR #5935⁵⁸⁹ - Fixing serialization to work with buffers of unsigned chars
- PR #5934⁵⁹⁰ - Attempting to fix datapar issues on CircleCI
- PR #5933⁵⁹¹ - Fix documentation for ranges algorithms
- PR #5932⁵⁹² - Remove malloc version constraint
- PR #5931⁵⁹³ - docs: fix & improve parallel algorithms documentation
- PR #5930⁵⁹⁴ - Add boost to hip builder
- PR #5929⁵⁹⁵ - Apply fixes to M1/MacOS related stack allocation to all relevant spots
- PR #5928⁵⁹⁶ - updated context_generic_context to accommodate arm64_arch_8/Apple architecture
- PR #5927⁵⁹⁷ - Public derivation for counting_semaphore_var
- PR #5926⁵⁹⁸ - Fix doxygen warnings when building documentation

⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5954>

⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5953>

⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5951>

⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5947>

⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5944>

⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5943>

⁵⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5942>

⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5941>

⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5940>

⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5939>

⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5938>

⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5937>

⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5936>

⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5935>

⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5934>

⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5933>

⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5932>

⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5931>

⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5930>

⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5929>

⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5928>

⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5927>

⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5926>

- PR #5923⁵⁹⁹ - Fixing git checkout to reflect latest version tag
- PR #5922⁶⁰⁰ - A couple of unrelated changes in support of implementing P1673
- PR #5920⁶⁰¹ - [P2300] enhancements: receiver_of, sender_of improvements
- PR #5917⁶⁰² - Fixing various ‘held lock while suspending’ problems
- PR #5916⁶⁰³ - Fix minor doxygen parsing typo
- PR #5915⁶⁰⁴ - docs: fix broken api algo links
- PR #5914⁶⁰⁵ - Remove CSS rules - update sphinx version
- PR #5911⁶⁰⁶ - Removed references to hpx::vector in comments
- PR #5909⁶⁰⁷ - Remove stuff which is defined in the header
- PR #5906⁶⁰⁸ - Use BUILD_SHARED_LIBS correctly
- PR #5905⁶⁰⁹ - Fix incorrect usage of generator expressions
- PR #5904⁶¹⁰ - Delete FindBZip2.cmake
- PR #5901⁶¹¹ - Fix #5900
- PR #5899⁶¹² - Replace PMR based version of small_vector
- PR #5897⁶¹³ - Add missing “”
- PR #5896⁶¹⁴ - Docs: Add serialization tutorial.
- PR #5895⁶¹⁵ - Update to V1.9.0 on master
- PR #5894⁶¹⁶ - Fix executor_with_thread_hooks example
- PR #5890⁶¹⁷ - Adding simple average power consumption performance counter
- PR #5889⁶¹⁸ - Par unseq/unseq adding
- PR #5888⁶¹⁹ - Support for data-parallelism for reduce, transform reduce, transform_binary_reduce algorithms
- PR #5887⁶²⁰ - Fixing dangling reference in serialization of non-default constructible types
- PR #5879⁶²¹ - New performance counters related to zero-copy chunks.

⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5923>

⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5922>

⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5920>

⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5917>

⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5916>

⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5915>

⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5914>

⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5911>

⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5909>

⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5906>

⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5905>

⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5904>

⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5901>

⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/5899>

⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5897>

⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5896>

⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5895>

⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5894>

⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5890>

⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5889>

⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5888>

⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5887>

⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5879>

2.10.3 HPX V1.8.0 (May 18, 2022)

With HPX parallel algorithms been fully adapted to C++20 the new release achieves full conformance with C++20 concurrency and parallelism facilities. HPX now supports all of the algorithms as specified by C++20. We have added support for vectorization to more of our algorithms. Much work has been done towards implementing P2300 (“`std::execution`”) and implementing the underlying senders/receivers facilities. Finally, The new release comes with a brand new documentation interface!

General changes

- The new documentation can now be found on our webpage: <https://hpx-docs.stellar-group.org>. This includes a completely new and user-friendly interface environment along with restructuring of certain components. The content in the “Quick start”, “Manual” and “Examples” was improved, while the “Build system” page was adapted to include necessary information for newcomers.
- With the vectorization support available in modern hardware architectures HPX now provides new data-parallel vector execution policies `hpx::execution::simd` and `hpx::execution::par_simd` that enable significant speed-up of our parallel algorithm implementations. The following algorithms now support SIMD execution:
 - `copy`, `copy_n`
 - `generate`
 - `adjacent_difference`, `adjacent_find`
 - `all_of`, `any_of`, `none_of`
 - `equal`, `mismatch`,
 - `inner_product`
 - `count`, `count_if`
 - `fill`, `fill_n`
 - `find`, `find_end`, `find_first_of`, `find_if`, `find_if_not`
 - `for_each`, `for_each_n`
 - `generate`, `generate_n`.
- Based on top of P2300 the HPX parallel algorithms now support the pipeline syntax towards an effort to unify their usage along with senders/receivers. The HPX parallel algorithms can now bind with senders/receivers using the pipeline operator.
- Several changes took place on the executors provided by HPX:
 - The executors now support the `num_cores` options in order for the user to be able to specify the desired number of cores to be used in the correspodning execution.
 - The `scheduler` executor was implemented on top of senders/receivers and can be used with all HPX facilities that schedule new work, such as parallel algorithms, `hpx::async`, `hpx::dataflow`, etc.
 - The performance of `fork_join_executor` was improved.
 - The following algorithms have been added/adapted to be C++20 conformant:
 - `min_element`
 - `max_element`
 - `minmax_element`
 - `starts_with`

- ends_with
- swap_ranges
- unique
- unique_copy
- rotate
- rotate_copy
- sort
- shift_left
- shift_right
- stable_sort
- partition
- partition_copy
- stable_partition
- adjacent_difference
- nth_element
- partial_sort
- partial_sort_copy.

- HPX_FORWARD/HPX_MOVE macros were introduced that replaced the `std::move` and `std::forward` facilities that in the library code.
- Hangs on distributed barrier were fixed.
- The performance of `scan_partitioner` was improved.
- Support was added for `thread_priority` to the `parallel_execution_policy`
- Regarding senders/receivers and the P2300 proposal various actions took place. `stop_token` was adapted to the recent proposal version (`in_place_stop_token` was introduced). Also hint, annotation, priority and stacksize properties were added to the scheduler executor. Stop support was added to `when_all`. Support for completion signatures was added. The following schedulers and algorithms were added:
 - `get_completion_scheduler`
 - `any_sender` and `unique_any_sender`
 - `split sender`
 - `transform_mpi sender`
 - `transfer sender`
 - `let_error`, `let_stopped`
 - `get_env` and related environment queries
 - `schedule`, `set_value`, `set_error`, `set_done`, `start` and `connect` are now proper customization points as defined in P2300.
- Several namespaces were altered towards conformance with C++20. Compatibility layers have been added and the old versions will be removed in next releases. The namespace changes are the following:
 - `hpx::parallel::induction/reduction` were moved into namespace `hpx::experimental`

- `for_loop` and friends were moved into namespace `hpx::experimental`.
 - `hpx::util::optional` and friends were moved into namespace `hpx`.
 - `hpx::lcos::barrier` has been moved into the `hpx::distributed` namespace and `hpx::lcos::local::cpp20_barrier` has been renamed to `barrier` and moved into the `hpx` namespace.
 - `hpx::lcos::latch` has been moved into the `hpx::distributed` namespace and `lcos::local::latch` has been moved into the `hpx` namespace. The `count_down_and_wait()` functionality of `latch` has been renamed to `arrive_and_wait()`.
 - `hpx::util::unique_function_nonser` has been renamed to `hpx::move_only_function`.
 - `hpx::util::unique_function` has been renamed to `hpx::distributed::move_only_function`.
 - `hpx::util::function` has been renamed to `hpx::distributed::function`.
 - `hpx::util::function_nonser` has been renamed to `hpx::function`.
 - `hpx::util::function_ref` have been moved to namespace `hpx`.
 - `hpx::lcos::split_future` changed namespace and is now used as `hpx::split_future`.
 - `hpx::lcos::local::counting_semaphore` has been deprecated and `hpx::lcos::local::cpp20_counting_semaphore` has been renamed to `hpx::counting_semaphore`.
 - `hpx::lcos::local::cpp20_binary_semaphore` has been renamed to `hpx::binary_semaphore`.
 - `hpx::lcos::local::sliding_semaphore` has been renamed to `hpx::sliding_semaphore` and
 - `hpx::lcos::local::sliding_semaphore_var` has been renamed to `hpx::sliding_semaphore_var`.
 - `hpx::lcos::local::spinlock` has been renamed to `hpx::spinlock`.
 - `hpx::lcos::local::mutex` has been renamed to `hpx::mutex`.
 - `hpx::lcos::local::timed_mutex` has been renamed to `hpx::timed_mutex`.
 - `hpx::lcos::local::no_mutex` has been renamed to `hpx::no_mutex`.
 - `hpx::lcos::local::recursive_mutex` has been renamed to `hpx::recursive_mutex`.
 - `hpx::lcos::local::shared_mutex` has been renamed to `hpx::shared_mutex`.
 - `hpx::lcos::local::upgrade_lock` has been renamed to `hpx::upgrade_lock`.
 - `hpx::lcos::local::upgrade_to_unique_lock` has been renamed to `hpx::upgrade_to_unique_lock`.
 - `hpx::lcos::local::condition_variable` has been renamed to `hpx::condition_variable`. `hpx::lcos::local::condition_variable_var` has been renamed to `hpx::condition_variable_var`.
 - `hpx::lcos::local::once_flag` has been renamed to `hpx::once_flag`, and . `hpx::lcos::local::call_once` has been renamed to `hpx::call_once`.
- The new LCI (Lightweight Communication Interface) parcelport was added that supports irregular and asynchronous applications like graph analysis, sparse linear algebra, modern parallel architectures etc. Major features include:
 - Support for advanced communication primitives like two sided send/recv and one sided remote put.
 - Better multi-threaded performance.

- Explicit user control of communication resource.
- Flexible signaling mechanisms (synchronizer, completion queue, active message handler).
- The following CMake flags were added, mostly to support using HPX as a backend for SHAD (<https://github.com/pnml/SHAD>). Please note that these options enable questionable functionalities, partially they even enable undefined behavior. Please only use any of them if you know what you're doing:
 - `HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION`
 - `HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE`
 - `HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS`

Breaking changes

- Minimum required C++ standard library is C++17.
- Support for GCC 7 and Clang 8.0.0 and below has been removed.
- CUDA version required updated to 11.4.
- CMake version required updated to 3.18.
- The default version of Asio used was updated to 1.20.0.
- The default version of APEX used was updated to 2.5.1.
- APEX version was updated to 2.5.1.
- `tagged_pair` and `tagged_tuple` were removed.
- `tag_dispatch` was renamed to `tag_invoke`.
- `hpx.max_backgroud_threads` was renamed to `hpx.parcel.max_background_threads`.
- The following CMake flags were removed after being deprecated for at least two releases:
 - `HPX_SCHEDULER_MAX_TERMINATED_THREADS`
 - `HPX_WITH_GOOGLE_PERFTOOLS`
 - `HPX_WITH_INIT_START_OVERLOADS_COMPATIBILITY`
 - `HPX_HAVE_{COROUTINE,PLUGIN}_GCC_HIDDEN_VISIBILITY`
 - `HPX_TOP_LEVEL`
 - `HPX_WITH_COMPUTE_CUDA`
 - `HPX_WITH_ASYNC_CUDA`
- `annotate_function` was renamed to `scoped_annotation`.
- `execution::transform` was renamed to `execution::then`.
- `execution::detach` was renamed to `execution::start_detached`.
- `execution::on_sender` was renamed to `execution::schedule_on`.
- `execution::just_on` was renamed to `execution::just_transfer`.
- `execution::set_done` was renamed to `execution::set_stopped`.

Closed issues

- Issue #5871⁶²² - distributed::channel.register_as terminates the active task.
- Issue #5856⁶²³ - Performance counters do not compile
- Issue #5828⁶²⁴ - hpx::distributed::barrier errors
- Issue #5812⁶²⁵ - OctoTiger does not compile with HPX master and CUDA 11.5
- Issue #5784⁶²⁶ - HPX failing with co_await and hpx::when_all(futures)
- Issue #5774⁶²⁷ - CMake can't find HPXCacheVariables.cmake
- Issue #5764⁶²⁸ - Fix HIP problem
- Issue #5724⁶²⁹ - Missing binary filter compression header
- Issue #5721⁶³⁰ - Cleanup after repository split
- Issue #5701⁶³¹ - It seems that the tcp parcelport is running, and the MPI parcelport is ignored
- Issue #5692⁶³² - Kokkos compilation fails when using both HPX and CUDA execution spaces with gcc 9.3.0
- Issue #5686⁶³³ - Rename *annotate_function*
- Issue #5668⁶³⁴ - HPX does not detect the C++ 20 standard using gcc 11.2
- Issue #5666⁶³⁵ - Compilation error using boost 1.76 and gcc 11.2.1
- Issue #5653⁶³⁶ - Implement P2248 for our algorithms
- Issue #5647⁶³⁷ - [User input needed] Remove (CUDA) compute functionality?
- Issue #5590⁶³⁸ - hello_world_distributed fails on startup with HPX stable, MPICH 3.3.2, on Deep Bayou
- Issue #5570⁶³⁹ - Rename tag_dispatch to tag_invoke
- Issue #5566⁶⁴⁰ - can't build simple example: "Cannot use the dummy implementation of future_then_dispatch"
- Issue #5565⁶⁴¹ - build failure: hpx::string_util::trim()
- Issue #5553⁶⁴² - Github action to validate the cff file refs #5471
- Issue #5504⁶⁴³ - CMake does not work for HPX 1.7.0 on Piz Daint

⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/5871>

⁶²³ <https://github.com/STELLAR-GROUP/hpx/issues/5856>

⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5828>

⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5812>

⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5784>

⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5774>

⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5764>

⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5724>

⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5721>

⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/5701>

⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/5692>

⁶³³ <https://github.com/STELLAR-GROUP/hpx/issues/5686>

⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5668>

⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5666>

⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5653>

⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5647>

⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5590>

⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5570>

⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5566>

⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/5565>

⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/5553>

⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/5504>

- Issue #5503⁶⁴⁴ - Use contiguous index queue in bulk execution to reduce number of spawned tasks
- Issue #5502⁶⁴⁵ - C++20 std::coroutine cmake detection
- Issue #5478⁶⁴⁶ - hpx.dll built with vcpkg got functions pointing to the same location
- Issue #5472⁶⁴⁷ - Compilation error with cuda/11.3
- Issue #5469⁶⁴⁸ - Compiler warning about HPX_NODISCARD when building with APEX
- Issue #5463⁶⁴⁹ - Address minor comments of the C++17 PR bump
- Issue #5456⁶⁵⁰ - Use *std::ranges::iter_swap* where available
- Issue #5404⁶⁵¹ - Build fails with error “Cannot open include file asio/io_context.hpp”
- Issue #5381⁶⁵² - Add starts_with and ends_with algorithms
- Issue #5344⁶⁵³ - Further simplify tag_invoke helpers
- Issue #5269⁶⁵⁴ - Allow setting a label on executors/policies
- Issue #5219⁶⁵⁵ - (Re-)Implement executor API on top of sender/receiver infrastructure
- Issue #5216⁶⁵⁶ - Performance counter module not loading
- Issue #5162⁶⁵⁷ - Require C++17 support
- Issue #5156⁶⁵⁸ - Disentangle segmented algorithms
- Issue #5118⁶⁵⁹ - Lock held while suspending
- Issue #5111⁶⁶⁰ - Tests fail to build with binary_filter plugins enabled
- Issue #5110⁶⁶¹ - Tests don't get built
- Issue #5105⁶⁶² - PAPI performance counters not available
- Issue #5002⁶⁶³ - hpx::lcos::barrier() results in deadlock
- Issue #4992⁶⁶⁴ - Clang-format the rest of the files
- Issue #4987⁶⁶⁵ - Use std::function in public APIs
- Issue #4871⁶⁶⁶ - HEP: conformance to C++20

⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5503>

⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5502>

⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5478>

⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5472>

⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5469>

⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5463>

⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5456>

⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/5404>

⁶⁵² <https://github.com/STELLAR-GROUP/hpx/issues/5381>

⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/5344>

⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5269>

⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5219>

⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5216>

⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5162>

⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5156>

⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5118>

⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5111>

⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/5110>

⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/5105>

⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/5002>

⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4992>

⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4987>

⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4871>

- Issue #4822⁶⁶⁷ - Adapt parallel algorithms to C++20
- Issue #4736⁶⁶⁸ - Deprecate hpx::flush and hpx::endl
- Issue #4558⁶⁶⁹ - Prevent work-stealing from stalling
- Issue #4495⁶⁷⁰ - Add anchor links to table rows in documentation
- Issue #4469⁶⁷¹ - New thread state: *pending_low*
- Issue #4321⁶⁷² - After the modularization the libfabric parcelport does not compile
- Issue #4308⁶⁷³ - Using APEX on multinode jobs when HPX_WITH_NETWORKING = OFF
- Issue #3995⁶⁷⁴ - Use C++20 std::source_location where available, adapt ours to conform
- Issue #3861⁶⁷⁵ - Selected processor does not support ‘yield’ in ARM mode
- Issue #3706⁶⁷⁶ - Add shift_left and shift_right algorithms
- Issue #3646⁶⁷⁷ - Parallel algorithms should accept iterator/sentinel pairs
- Issue #3636⁶⁷⁸ - HPX Modularization
- Issue #3546⁶⁷⁹ - Modularization of HPX
- Issue #3474⁶⁸⁰ - Modernize CMake used in HPX
- Issue #1836⁶⁸¹ - hpx::parallel does not have a sort implementation
- Issue #1668⁶⁸² - Adapt all parallel algorithms to Ranges TS
- Issue #1141⁶⁸³ - Implement N4409 on top of HPX

Closed pull requests

- PR #5885⁶⁸⁴ - Testing newer ASIO version
- PR #5884⁶⁸⁵ - Fix miscellaneous doc sections
- PR #5882⁶⁸⁶ - Fixing OctoTiger incompatibility introduced recently
- PR #5881⁶⁸⁷ - Fixing recent patch that disables ATOMIC_FLAG_INIT for C++20 and up
- PR #5880⁶⁸⁸ - refactor: convert *counter_status* enum to enum class

⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4822>

⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4736>

⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4558>

⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4495>

⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/4469>

⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/4321>

⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/4308>

674 <https://github.com/STELLAR-GROUP/hpx/issues/3995>675 <https://github.com/STELLAR-GROUP/hpx/issues/3861>676 <https://github.com/STELLAR-GROUP/hpx/issues/3706>677 <https://github.com/STELLAR-GROUP/hpx/issues/3646>678 <https://github.com/STELLAR-GROUP/hpx/issues/3636>679 <https://github.com/STELLAR-GROUP/hpx/issues/3546>680 <https://github.com/STELLAR-GROUP/hpx/issues/3474>681 <https://github.com/STELLAR-GROUP/hpx/issues/1836>682 <https://github.com/STELLAR-GROUP/hpx/issues/1668>683 <https://github.com/STELLAR-GROUP/hpx/issues/1141>684 <https://github.com/STELLAR-GROUP/hpx/pull/5885>685 <https://github.com/STELLAR-GROUP/hpx/pull/5884>686 <https://github.com/STELLAR-GROUP/hpx/pull/5882>687 <https://github.com/STELLAR-GROUP/hpx/pull/5881>688 <https://github.com/STELLAR-GROUP/hpx/pull/5880>

- PR #5878⁶⁸⁹ - Docs: Replaced non-existent create_reducer function with create_communicator
- PR #5877⁶⁹⁰ - Doc updates hpx runtime and resources
- PR #5876⁶⁹¹ - Updates to documentation; grammar edits.
- PR #5875⁶⁹² - Doc updates starting the hpx runtime
- PR #5874⁶⁹³ - Doc updates launching configuring
- PR #5873⁶⁹⁴ - Prevent certain generated files from being deleted on reconfigure
- PR #5870⁶⁹⁵ - Adding support for the PJM batch environment
- PR #5867⁶⁹⁶ - Update CMakeLists.txt
- PR #5866⁶⁹⁷ - add cmake option HPX_WITH_PARCELPORT_COUNTERS
- PR #5864⁶⁹⁸ - ATOMIC_INIT_FLAG is deprecated starting C++20
- PR #5863⁶⁹⁹ - Adding llvm 14.0.0 with boost 1.79.0 to Jenkins
- PR #5861⁷⁰⁰ - Let install step proceed on CircleCI even if the segmented algorithms fail
- PR #5860⁷⁰¹ - Updating APEX tag
- PR #5859⁷⁰² - Splitting documentation generation steps on CircleCI
- PR #5854⁷⁰³ - Fixing left-overs from changing counter_type to enum class
- PR #5853⁷⁰⁴ - Adding HPX dependency tool (adapted from Boostdep tool)
- PR #5852⁷⁰⁵ - Optimize LCI parcelport
- PR #5851⁷⁰⁶ - Forking dynamic_bitset from Boost
- PR #5850⁷⁰⁷ - Convert perf_counters::counter_type enum to enum class.
- PR #5849⁷⁰⁸ - Update LCI parcelport to LCI v1.7.1
- PR #5848⁷⁰⁹ - Fedora related fixes
- PR #5847⁷¹⁰ - Fix API, troubleshooting & people
- PR #5844⁷¹¹ - Attempting to fix timeouts of segmented iterator tests

⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5878>

⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5877>

⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5876>

⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5875>

⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5874>

⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5873>

695 <https://github.com/STELLAR-GROUP/hpx/pull/5870>696 <https://github.com/STELLAR-GROUP/hpx/pull/5867>697 <https://github.com/STELLAR-GROUP/hpx/pull/5866>698 <https://github.com/STELLAR-GROUP/hpx/pull/5864>699 <https://github.com/STELLAR-GROUP/hpx/pull/5863>700 <https://github.com/STELLAR-GROUP/hpx/pull/5861>701 <https://github.com/STELLAR-GROUP/hpx/pull/5860>702 <https://github.com/STELLAR-GROUP/hpx/pull/5859>703 <https://github.com/STELLAR-GROUP/hpx/pull/5854>704 <https://github.com/STELLAR-GROUP/hpx/pull/5853>705 <https://github.com/STELLAR-GROUP/hpx/pull/5852>706 <https://github.com/STELLAR-GROUP/hpx/pull/5851>707 <https://github.com/STELLAR-GROUP/hpx/pull/5850>708 <https://github.com/STELLAR-GROUP/hpx/pull/5849>709 <https://github.com/STELLAR-GROUP/hpx/pull/5848>710 <https://github.com/STELLAR-GROUP/hpx/pull/5847>711 <https://github.com/STELLAR-GROUP/hpx/pull/5844>

- PR #5842⁷¹² - change the default value of HPX_WITH_LCI_TAG to v1.7
- PR #5841⁷¹³ - Move the split_future facilities into the namespace hpx
- PR #5840⁷¹⁴ - wait_xxx_nothrow functions return whether one of the futures is exceptional
- PR #5839⁷¹⁵ - Moving a list of synchronization primitives into namespace hpx
- PR #5837⁷¹⁶ - Moving latch types to hpx and hpx::distributed namespaces
- PR #5835⁷¹⁷ - Add missing compatibility layer for id_type::management_type values
- PR #5834⁷¹⁸ - API docs changes
- PR #5831⁷¹⁹ - Further improvement actions to rotate
- PR #5830⁷²⁰ - Exposing zero-copy serialization threshold through configuration option
- PR #5829⁷²¹ - Attempting to fix failing barrier test
- PR #5827⁷²² - Add back explicit template parameter to *ignore_while_checking* to compile with nvcc
- PR #5826⁷²³ - Reduce number of allocations while calling async_bulk_execute
- PR #5825⁷²⁴ - Steal from neighboring NUMA domain only
- PR #5823⁷²⁵ - Remove obsolete directories and adjust build system
- PR #5822⁷²⁶ - Clang-format remaining files
- PR #5821⁷²⁷ - Enable permissive- flag on Windows GitHub actions builders
- PR #5820⁷²⁸ - Convert throwmode enum to enum class
- PR #5819⁷²⁹ - Marking customization points for intrusive_ptr as noexcept
- PR #5818⁷³⁰ - Unconditionally use C++17 attributes
- PR #5817⁷³¹ - Modernize naming modules
- PR #5816⁷³² - Modernize cache module
- PR #5815⁷³³ - Reapply flyby changes from #5467
- PR #5814⁷³⁴ - Avoid test timeouts by reducing test sizes

⁷¹² <https://github.com/STELLAR-GROUP/hpx/pull/5842>

⁷¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5841>

⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5840>

⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5839>

⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5837>

⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5835>

⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5834>

⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5831>

⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5830>

⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5829>

⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/5827>

⁷²³ <https://github.com/STELLAR-GROUP/hpx/pull/5826>

⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5825>

⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5823>

⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5822>

⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5821>

⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5820>

⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5819>

⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5818>

⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5817>

⁷³² <https://github.com/STELLAR-GROUP/hpx/pull/5816>

⁷³³ <https://github.com/STELLAR-GROUP/hpx/pull/5815>

⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5814>

- PR #5813⁷³⁵ - The CUDA problem is not fixed in V11.5 yet...
- PR #5811⁷³⁶ - Make sure reduction value is properly moved, when possible
- PR #5810⁷³⁷ - Improve error reporting during device initialization in HIP environments
- PR #5809⁷³⁸ - Converting scheduler enums into enum class
- PR #5808⁷³⁹ - Deprecate hpx::flush and friends
- PR #5807⁷⁴⁰ - Use C++20 std::source_location, if available
- PR #5806⁷⁴¹ - Moving promise and packaged_task to new namespaces
- PR #5805⁷⁴² - Attempting to fix a test failure when using the LCI parcelpor
- PR #5803⁷⁴³ - Attempt to fix CUDA related OctoTiger problems
- PR #5800⁷⁴⁴ - Add option to restrict MPI background work to subset of cores
- PR #5798⁷⁴⁵ - Adding MPI as a dependency to APEX
- PR #5797⁷⁴⁶ - Extend Sphinx role to support arbitrary text to display on a link
- PR #5796⁷⁴⁷ - Disable CUDA tests that cause NVCC to silently fail without error messages
- PR #5795⁷⁴⁸ - Avoid writing path and directories into HPXCacheVariables.cmake
- PR #5793⁷⁴⁹ - Remove features that are deprecated since V1.6
- PR #5792⁷⁵⁰ - Making sure num_cores is properly handled by parallel_executor
- PR #5791⁷⁵¹ - Moving bind, bind_front, bind_back to namespace hpx
- PR #5790⁷⁵² - Moving serializable function/move_only_function into namespace hpx::distributed
- PR #5787⁷⁵³ - Remove unneeded (and commented) tests
- PR #5786⁷⁵⁴ - Attempting to fix hangs in distributed barrier
- PR #5785⁷⁵⁵ - add cmake code to detect arm64 on macOS
- PR #5783⁷⁵⁶ - Moving function and function_ref into namespace hpx
- PR #5781⁷⁵⁷ - Updating used version of Visual Studio

⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5813>

⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5811>

⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5810>

⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5809>

⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5808>

⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5807>

⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5806>

⁷⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5805>

⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5803>

⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5800>

⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5798>

⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5797>

⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5796>

⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5795>

749 <https://github.com/STELLAR-GROUP/hpx/pull/5793>750 <https://github.com/STELLAR-GROUP/hpx/pull/5792>751 <https://github.com/STELLAR-GROUP/hpx/pull/5791>752 <https://github.com/STELLAR-GROUP/hpx/pull/5790>753 <https://github.com/STELLAR-GROUP/hpx/pull/5787>754 <https://github.com/STELLAR-GROUP/hpx/pull/5786>755 <https://github.com/STELLAR-GROUP/hpx/pull/5785>756 <https://github.com/STELLAR-GROUP/hpx/pull/5783>757 <https://github.com/STELLAR-GROUP/hpx/pull/5781>

- PR #5780⁷⁵⁸ - Update Piz Daint Jenkins configurations from gcc/clang 7 to 8
- PR #5778⁷⁵⁹ - Updated for_loop.hpp
- PR #5777⁷⁶⁰ - Update reference for foreach benchmark
- PR #5775⁷⁶¹ - Move optional into namespace hpx
- PR #5773⁷⁶² - Moving barrier to consolidated namespaces
- PR #5772⁷⁶³ - Adding missing docs for ranges::find_if and find_if_not algorithms
- PR #5771⁷⁶⁴ - Moving for_loop into namespace hpx::experimental
- PR #5770⁷⁶⁵ - Fixing HIP issues
- PR #5769⁷⁶⁶ - Slight improvement of small_vector performance
- PR #5766⁷⁶⁷ - Fixing a integral conversion warning
- PR #5765⁷⁶⁸ - Adding a sphinx role allowing to link to a file directly in github
- PR #5763⁷⁶⁹ - add num_cores facility
- PR #5762⁷⁷⁰ - Fix Public API main page
- PR #5761⁷⁷¹ - Add missing inline to mpi_exception.hpp error_message function
- PR #5760⁷⁷² - Update cdash build url
- PR #5759⁷⁷³ - Switch to use generic rostam SLURM partitions
- PR #5758⁷⁷⁴ - Adding support for P2300 completion signatures
- PR #5757⁷⁷⁵ - Fix missing links in Public API
- PR #5756⁷⁷⁶ - Add stop support to when_all
- PR #5755⁷⁷⁷ - Support for data-parallelism for mismatch algorithm
- PR #5754⁷⁷⁸ - Support for data-parallelism for equal algorithm
- PR #5751⁷⁷⁹ - Propagate MPI dependencies to command line handling
- PR #5750⁷⁸⁰ - Make sure required MPI initialization flags are properly applied and supported

⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5780>

⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5778>

⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5777>

⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5775>

⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5773>

⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5772>

⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5771>

⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5770>

⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5769>

⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5766>

⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5765>

⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5763>

⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5762>

⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/5761>

⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/5760>

⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/5759>

⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5758>

⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5757>

⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5756>

⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5755>

⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5754>

⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5751>

⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5750>

- PR #5749⁷⁸¹ - P2300 stop token
- PR #5748⁷⁸² - Adding environmental query CPOs
- PR #5747⁷⁸³ - Renaming set_done to set_stopped (as per P2300)
- PR #5745⁷⁸⁴ - Modernize serialization module
- PR #5743⁷⁸⁵ - Add check for MPICH and set the correct env to support multi-threaded
- PR #5742⁷⁸⁶ - Remove obsolete files related to cupid, etc.
- PR #5741⁷⁸⁷ - Support for data-parallelism for adjacent find
- PR #5740⁷⁸⁸ - Support for data-parallelism for find algorithms
- PR #5739⁷⁸⁹ - Enable the option to attach a debugger on a segmentation fault (linux)
- PR #5738⁷⁹⁰ - Fixing spell-checking errors
- PR #5737⁷⁹¹ - Attempt to fix migrate_component issue
- PR #5736⁷⁹² - Set commit status from Jenkins also for special branches
- PR #5734⁷⁹³ - Revert #5586
- PR #5732⁷⁹⁴ - Attempt to improve build-id reporting to cdash
- PR #5731⁷⁹⁵ - Randomly delay execution of bash scripts launched by Jenkins
- PR #5729⁷⁹⁶ - Workaround for CMake/Ninja generator OOM problem
- PR #5727⁷⁹⁷ - Moving compression plugins to components directory
- PR #5726⁷⁹⁸ - Moving/consolidating parcel coalescing plugin sources
- PR #5725⁷⁹⁹ - Making sure headers for serialization filters are being installed
- PR #5723⁸⁰⁰ - Moving more tests to modules
- PR #5722⁸⁰¹ - Removing superfluous semicolons
- PR #5720⁸⁰² - Moving parcelports into modules
- PR #5719⁸⁰³ - Moving more files to parcelset module

⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5749>

⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5748>

⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5747>

⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5745>

⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5743>

⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5742>

⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5741>

⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5740>

⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5739>

⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5738>

⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5737>

⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5736>

⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5734>

⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5732>

⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5731>

⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5729>

⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5727>

⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5726>

⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5725>

⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5723>

⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5722>

⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5720>

⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5719>

- PR #5718⁸⁰⁴ - build: refactor sphinx config file
- PR #5717⁸⁰⁵ - Creating parcelset modules
- PR #5716⁸⁰⁶ - Avoid duplicate definition error
- PR #5715⁸⁰⁷ - The new LCI parcelport for HPX
- PR #5714⁸⁰⁸ - Refine propagation of **HPX_WITH_...** options
- PR #5713⁸⁰⁹ - Significantly reduce CI jobs run on Piz Daint
- PR #5712⁸¹⁰ - Updating jenkins configuration for Rostam2.2
- PR #5711⁸¹¹ - Refactor manual sections
- PR #5710⁸¹² - Making task_group serializable
- PR #5709⁸¹³ - Update the MPI cmake setup
- PR #5707⁸¹⁴ - Better diagnose parcel bootstrap problems
- PR #5704⁸¹⁵ - Test with hwloc 2.7.0 with GCC 11
- PR #5703⁸¹⁶ - Fix *counting_iterator* container tests
- PR #5702⁸¹⁷ - Attempting to fix CircleCI timeouts
- PR #5699⁸¹⁸ - Update CI to use Boost 1.78.0
- PR #5697⁸¹⁹ - Adding fork_join_executor to foreach_benchmark
- PR #5696⁸²⁰ - Modernize when_all and friends (when_any, when_some, when_each)
- PR #5693⁸²¹ - Fix test errors with *_GLIBCXX_DEBUG* defined
- PR #5691⁸²² - Rename *annotate_function* to *scoped_annotation*
- PR #5690⁸²³ - Replace tag_dispatch with tag_invoke in minmax segmented
- PR #5688⁸²⁴ - Remove more deprecated macros
- PR #5687⁸²⁵ - Add most important CMake options
- PR #5685⁸²⁶ - Fix future API

⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5718>

⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5717>

⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5716>

⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5715>

⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5714>

⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5713>

⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5712>

⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5711>

⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/5710>

⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5709>

⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5707>

⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5704>

⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5703>

⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5702>

⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5699>

⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5697>

⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5696>

⁸²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5693>

⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/5691>

⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/5690>

⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5688>

⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5687>

⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5685>

- PR #5684⁸²⁷ - Move lock registration to separate module and remove global lock registration
- PR #5683⁸²⁸ - Make hpx::wait_all etc. throw exceptions when waited futures hold exceptions and deprecate hpx::lcos::wait_all[_n] in favor of hpx::wait_all[_n]
- PR #5682⁸²⁹ - Fix macOS test exceptions
- PR #5681⁸³⁰ - docs: add links to hpx recepies
- PR #5680⁸³¹ - Embed base execution policies to datapar execution policies
- PR #5679⁸³² - Fix *fork_join_executor* with dynamic schedule
- PR #5678⁸³³ - Fix compilation of service executors with nvcc
- PR #5677⁸³⁴ - Remove compute_cuda module
- PR #5676⁸³⁵ - Don't require up-to-date approvals for bors
- PR #5675⁸³⁶ - Add default template type parameters for algorithms
- PR #5674⁸³⁷ - Allow using *any_sender* in global variables
- PR #5671⁸³⁸ - Making sure task_group can be reused
- PR #5670⁸³⁹ - Relax constraints on *execution::when_all*
- PR #5669⁸⁴⁰ - Use HPX_WITH_CXX_STANDARD for controlling C++ version
- PR #5667⁸⁴¹ - Attempt to fix compilation issues with Boost V1.76
- PR #5664⁸⁴² - Change logging errors to warnings in schedulers
- PR #5663⁸⁴³ - Use dynamic bitsets by default for CPU masks
- PR #5662⁸⁴⁴ - Disambiguate namespace for MSVC
- PR #5660⁸⁴⁵ - Replacing remaining std::forward and std::move with HPX_FORWARD and HPX_MOVE
- PR #5659⁸⁴⁶ - Modernize hpx::future and related facilities
- PR #5658⁸⁴⁷ - Replace HPX_INLINE_CONSTEXPR_VARIABLE with inline constexpr
- PR #5657⁸⁴⁸ - Remove tagged, tagged_pair and tagged_tuple, remove tuple/pair specializations
- PR #5656⁸⁴⁹ - Rename on execution::schedule_from, rename just_on to just_transfer, and add transfer

⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5684>

⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5683>

⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5682>

⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5681>

⁸³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5680>

⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/5679>

⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/5678>

⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5677>

⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5676>

⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5675>

⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5674>

⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5671>

⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5670>

⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5669>

⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5667>

⁸⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5664>

⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5663>

⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5662>

⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5660>

⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5659>

⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5658>

⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5657>

⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5656>

- PR #5655⁸⁵⁰ - Avoid for module lists to grow indefinitely in cmake cache
- PR #5649⁸⁵¹ - build: replace usage of Python's reserved words and functions as variable names
- PR #5648⁸⁵² - Modernize action modules and related code
- PR #5646⁸⁵³ - Fix ends_with test
- PR #5645⁸⁵⁴ - Add matrix multiplication example
- PR #5644⁸⁵⁵ - Rename execution::transform to execution::then and execution::detach to execution::start_detached
- PR #5643⁸⁵⁶ - Update performance test references
- PR #5642⁸⁵⁷ - Adapting adjacent_difference to work with proxy iterators
- PR #5641⁸⁵⁸ - Factorize perftests scripts
- PR #5640⁸⁵⁹ - Fixed links to sources in Sphinx documentation
- PR #5639⁸⁶⁰ - Fix generate datapar tests for Vc
- PR #5638⁸⁶¹ - Simd all any none
- PR #5637⁸⁶² - Use bors for merging pull requests
- PR #5636⁸⁶³ - Fix leftover std::holds_alternative usage
- PR #5635⁸⁶⁴ - Update container image tag in GitHub actions HIP configuration
- PR #5633⁸⁶⁵ - Moving packaged_task to module futures
- PR #5632⁸⁶⁶ - Tell Asio to use std::aligned_new only if available
- PR #5631⁸⁶⁷ - Adding tag parameter to channel communicator get/set
- PR #5630⁸⁶⁸ - Add partial_sort_copy and adapt partial sort to c++ 20
- PR #5629⁸⁶⁹ - Set HPX_WITH_FETCH_ASIO to OFF as available in the docker image
- PR #5628⁸⁷⁰ - Add Clang 13 CI configuration
- PR #5627⁸⁷¹ - Replace alternative keyword
- PR #5626⁸⁷² - docs: add support for BibTeX references in Sphinx docs

⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5655>

⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/5649>

⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/5648>

⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/5646>

⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5645>

⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5644>

⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5643>

⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5642>

⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5641>

⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5640>

⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5639>

⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5638>

⁸⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5637>

⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5636>

⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5635>

⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5633>

⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5632>

⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5631>

⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5630>

⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5629>

⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5628>

⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/5627>

⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/5626>

- PR #5624⁸⁷³ - Fix pkgconfig replacements involving CMAKE_INSTALL_PREFIX
- PR #5623⁸⁷⁴ - build: remove unused import from conf.py.in
- PR #5622⁸⁷⁵ - Remove HPX_WITH_VCPKG CMake option
- PR #5621⁸⁷⁶ - Replacing boost::container::small_vector
- PR #5620⁸⁷⁷ - Update Asio tag from 1.18.2 to 1.20.0
- PR #5619⁸⁷⁸ - Fix block_os_threads_1036 test
- PR #5618⁸⁷⁹ - Make sure condition variables are notified under a lock in the thread_pool_scheduler test
- PR #5617⁸⁸⁰ - Use advance_and_get_distance where required
- PR #5616⁸⁸¹ - Remove separately building segmented algorithms on CircleCI
- PR #5613⁸⁸² - Fix Vc datapar adjacent_difference
- PR #5609⁸⁸³ - docs: add anchor links to performance counter tables
- PR #5608⁸⁸⁴ - Fix header test error by adding missing numeric
- PR #5607⁸⁸⁵ - Fix simd adj diff
- PR #5605⁸⁸⁶ - Fix usage of HPX_INVOKE macro
- PR #5604⁸⁸⁷ - Make use of shell-session to allow non-copyable \$
- PR #5603⁸⁸⁸ - Suppress some MSVC warnings in C++20 mode
- PR #5602⁸⁸⁹ - Test HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE=OFF to one CI configuration
- PR #5601⁸⁹⁰ - Test case for any_sender should use hpx::tuple
- PR #5600⁸⁹¹ - Rename tag_dispatch back to tag_invoke
- PR #5599⁸⁹² - Change theme, fix Quickstart & Examples
- PR #5596⁸⁹³ - Use precompiled headers in tests
- PR #5595⁸⁹⁴ - Drop semicolons for macro calls
- PR #5594⁸⁹⁵ - Adapt datapar generate

⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/5624>

⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5623>

⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5622>

⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5621>

⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5620>

⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5619>

⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5618>

⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5617>

⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5616>

⁸⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5613>

⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5609>

⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5608>

⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5607>

⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5605>

887 <https://github.com/STELLAR-GROUP/hpx/pull/5604>888 <https://github.com/STELLAR-GROUP/hpx/pull/5603>889 <https://github.com/STELLAR-GROUP/hpx/pull/5602>890 <https://github.com/STELLAR-GROUP/hpx/pull/5601>891 <https://github.com/STELLAR-GROUP/hpx/pull/5600>892 <https://github.com/STELLAR-GROUP/hpx/pull/5599>893 <https://github.com/STELLAR-GROUP/hpx/pull/5596>894 <https://github.com/STELLAR-GROUP/hpx/pull/5595>895 <https://github.com/STELLAR-GROUP/hpx/pull/5594>

- PR #5593⁸⁹⁶ - Update any_sender to use tag_dispatch for execution customizations
- PR #5592⁸⁹⁷ - Add nth_element
- PR #5591⁸⁹⁸ - Remove unnecessary checks for C++17 for tests
- PR #5589⁸⁹⁹ - Add HPX_FORWARD/HPX_MOVE macros
- PR #5588⁹⁰⁰ - Fixing the output formatting for id_types
- PR #5586⁹⁰¹ - Remove local functionality
- PR #5585⁹⁰² - Delete GitExternal.cmake
- PR #5584⁹⁰³ - Serialization of hpx::tuple must use hpx::get
- PR #5583⁹⁰⁴ - fix coroutine_traits allocate calls, add unhandled_exception() implementation.
- PR #5582⁹⁰⁵ - Make more examples work with local runtime
- PR #5581⁹⁰⁶ - Add support for several performance tests in CI
- PR #5580⁹⁰⁷ - Adapt simd adj diff
- PR #5579⁹⁰⁸ - Split absolute paths for generated pkg-config files into -L/-l parts
- PR #5577⁹⁰⁹ - fix unit fill test for datapar with Vc
- PR #5576⁹¹⁰ - Update forgotten ‘Full’ names
- PR #5575⁹¹¹ - Change scan partitioner implementation
- PR #5574⁹¹² - Remove a few deprecated and unused CMake options
- PR #5572⁹¹³ - Remove more guards for the distributed runtime
- PR #5571⁹¹⁴ - Add workaround for libstdc++ in string_util trim
- PR #5569⁹¹⁵ - Use no_unique_address in sender adaptors
- PR #5568⁹¹⁶ - Change try catch block to try_catch_exception_ptr
- PR #5567⁹¹⁷ - Make default_agent::yield actually yield
- PR #5564⁹¹⁸ - Adjacent

⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5593>

⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5592>

⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5591>

⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5589>

⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5588>

⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5586>

⁹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5585>

⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5584>

⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5583>

⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5582>

⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5581>

⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5580>

⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5579>

⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5577>

⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5576>

⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5575>

⁹¹² <https://github.com/STELLAR-GROUP/hpx/pull/5574>

⁹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5572>

⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5571>

⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5569>

⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5568>

⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5567>

⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5564>

- PR #5562⁹¹⁹ - More changes to overcome build problems on Windows after recent module rearrangements
- PR #5560⁹²⁰ - Update tests and examples
- PR #5559⁹²¹ - Fixing cmake folder names after module restructuring
- PR #5558⁹²² - Fixing wrong module dependencies
- PR #5557⁹²³ - Adding an example for the new channel_communicator API
- PR #5556⁹²⁴ - Remove leftover thread pool os executor tests
- PR #5555⁹²⁵ - Add option enabling serializing raw pointers
- PR #5554⁹²⁶ - Make sure command line aliasing is properly handled
- PR #5552⁹²⁷ - Modernizing some of the async facilities
- PR #5551⁹²⁸ - Fixing for local executions of actions to properly set task names
- PR #5550⁹²⁹ - Update CUDA module in clang-cuda configuration
- PR #5549⁹³⁰ - Fixing agent_ref::yield_k to actually call yield_k
- PR #5548⁹³¹ - Making get_action_name() noexcept
- PR #5547⁹³² - Fixing communication set
- PR #5546⁹³³ - Fixing shutdown problems caused by missing ref-counting
- PR #5545⁹³⁴ - Remove wrong move in thread_pool_scheduler_bulk.hpp
- PR #5543⁹³⁵ - Extend launch policy to carry stack size and scheduling hint in addition to priority
- PR #5542⁹³⁶ - Simplify execution CPOs
- PR #5540⁹³⁷ - Adapt partition, partition_copy and stable_partition to C++ 20
- PR #5539⁹³⁸ - Adapt mismatch to support sentinels
- PR #5538⁹³⁹ - Document specific sphinx version required for the documentation
- PR #5537⁹⁴⁰ - Test release and debug builds on Piz Daint
- PR #5536⁹⁴¹ - This fixes referencing stale iterators during the execution of binary mismatch

⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5562>

⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5560>

⁹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5559>

⁹²² <https://github.com/STELLAR-GROUP/hpx/pull/5558>

⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/5557>

⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5556>

⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5555>

⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5554>

⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5552>

⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5551>

⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5550>

⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5549>

⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5548>

⁹³² <https://github.com/STELLAR-GROUP/hpx/pull/5547>

933 <https://github.com/STELLAR-GROUP/hpx/pull/5546>934 <https://github.com/STELLAR-GROUP/hpx/pull/5545>935 <https://github.com/STELLAR-GROUP/hpx/pull/5543>936 <https://github.com/STELLAR-GROUP/hpx/pull/5542>937 <https://github.com/STELLAR-GROUP/hpx/pull/5540>938 <https://github.com/STELLAR-GROUP/hpx/pull/5539>939 <https://github.com/STELLAR-GROUP/hpx/pull/5538>940 <https://github.com/STELLAR-GROUP/hpx/pull/5537>941 <https://github.com/STELLAR-GROUP/hpx/pull/5536>

- PR #5535⁹⁴² - Rename simdpar to par_simd
- PR #5534⁹⁴³ - Fix Quick start & Manual Docs
- PR #5533⁹⁴⁴ - Fix *annotate_function* for *std::string*
- PR #5532⁹⁴⁵ - Update two remaining apex links from khuck to UO-OACISS
- PR #5531⁹⁴⁶ - Use contiguous_index_queue in thread_pool_scheduler
- PR #5530⁹⁴⁷ - Eagerly initialize a configurable number of threads on scheduler/thread queue init
- PR #5529⁹⁴⁸ - Update benchmarks and add support for scheduler_executor
- PR #5528⁹⁴⁹ - Add missing properties to executors/schedulers
- PR #5527⁹⁵⁰ - Set local thread/pool number in local/static_queue_scheduler
- PR #5526⁹⁵¹ - Update Rostam HIP configuration to use 4.3.0
- PR #5525⁹⁵² - Fix Building HPX in Quick start
- PR #5524⁹⁵³ - Upload image on cdash
- PR #5523⁹⁵⁴ - Modernize facilities related to hpx::sync
- PR #5522⁹⁵⁵ - Add sender overloads for remaining algorithms
- PR #5521⁹⁵⁶ - Minor changes that improve performance
- PR #5520⁹⁵⁷ - Update reference as perftests failing regularly
- PR #5519⁹⁵⁸ - Add transform_mpi sender adapter
- PR #5518⁹⁵⁹ - Add sender overloads to rotate/rotate_copy
- PR #5517⁹⁶⁰ - Fix coroutine integration
- PR #5515⁹⁶¹ - Avoid deadlock in ignore_while_locked_1485 test
- PR #5514⁹⁶² - Add split sender adapter
- PR #5512⁹⁶³ - Update Rostam HIP configuration
- PR #5511⁹⁶⁴ - Fix Asio target name for precompiled headers

⁹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5535>

⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5534>

⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5533>

⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5532>

⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5531>

⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5530>

⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5529>

⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5528>

⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5527>

⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/5526>

⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/5525>

⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/5524>

⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5523>

⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5522>

⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5521>

⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5520>

⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5519>

⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5518>

⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5517>

⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5515>

⁹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5514>

⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5512>

⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5511>

- PR #5510⁹⁶⁵ - Add any_sender and unique_any_sender
- PR #5509⁹⁶⁶ - Test with Boost 1.77 on gcc/clang-newest configurations
- PR #5508⁹⁶⁷ - Minor release changes from 1.7.1
- PR #5507⁹⁶⁸ - Add missing commits from scheduler_executor PR
- PR #5506⁹⁶⁹ - Fix condition for checking if we should use our own variant
- PR #5501⁹⁷⁰ - Attempt to fix thread_pool_scheduler test
- PR #5493⁹⁷¹ - Update Jenkins GitHub token to use StellarBot GitHub account
- PR #5490⁹⁷² - Fix clang-format error on master
- PR #5487⁹⁷³ - Add get_completion_scheduler CPO and customize bulk for thread_pool_scheduler
- PR #5484⁹⁷⁴ - Add missing header to jacobi_component/server/solver.hpp
- PR #5481⁹⁷⁵ - Changing the APEX repository to the new location
- PR #5479⁹⁷⁶ - Fix version check for CUDA noexcept/result_of bug
- PR #5477⁹⁷⁷ - Require cxx17 minor comments
- PR #5476⁹⁷⁸ - Fix cmake format error
- PR #5475⁹⁷⁹ - Require CMake 3.18 as it is already a requirement for CUDA
- PR #5474⁹⁸⁰ - Make the cuda parameters of try_compile optional
- PR #5473⁹⁸¹ - Update cuda arch and change cuda version
- PR #5471⁹⁸² - Add corrected citation.cff
- PR #5470⁹⁸³ - Adapt stable_sort to C++ 20
- PR #5468⁹⁸⁴ - Experimentation to make the perftest report public
- PR #5466⁹⁸⁵ - Add shift_left and shift_right algorithms
- PR #5465⁹⁸⁶ - Adapt datapar fill
- PR #5464⁹⁸⁷ - Moving tag_dispatch to separate module

⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5510>

⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5509>

⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5508>

⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5507>

⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5506>

⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5501>

971 <https://github.com/STELLAR-GROUP/hpx/pull/5493>972 <https://github.com/STELLAR-GROUP/hpx/pull/5490>973 <https://github.com/STELLAR-GROUP/hpx/pull/5487>974 <https://github.com/STELLAR-GROUP/hpx/pull/5484>975 <https://github.com/STELLAR-GROUP/hpx/pull/5481>976 <https://github.com/STELLAR-GROUP/hpx/pull/5479>977 <https://github.com/STELLAR-GROUP/hpx/pull/5477>978 <https://github.com/STELLAR-GROUP/hpx/pull/5476>979 <https://github.com/STELLAR-GROUP/hpx/pull/5475>980 <https://github.com/STELLAR-GROUP/hpx/pull/5474>981 <https://github.com/STELLAR-GROUP/hpx/pull/5473>982 <https://github.com/STELLAR-GROUP/hpx/pull/5471>983 <https://github.com/STELLAR-GROUP/hpx/pull/5470>984 <https://github.com/STELLAR-GROUP/hpx/pull/5468>985 <https://github.com/STELLAR-GROUP/hpx/pull/5466>986 <https://github.com/STELLAR-GROUP/hpx/pull/5465>987 <https://github.com/STELLAR-GROUP/hpx/pull/5464>

- PR #5461⁹⁸⁸ - Rename HPX_WITH_CUDA_COMPUTE with HPX_WITH_COMPUTE_CUDA
- PR #5460⁹⁸⁹ - Adapt sort to C++ 20
- PR #5459⁹⁹⁰ - Adapt rotate/rotate_copy to C++20
- PR #5458⁹⁹¹ - Adapt unique and unique_copy to C++ 20
- PR #5455⁹⁹² - Remove and clean up fallback sender implementations
- PR #5454⁹⁹³ - Make performance plot show even if similar performance
- PR #5453⁹⁹⁴ - Post 1.7.0 version bump
- PR #5452⁹⁹⁵ - Fix find_end parallel overload
- PR #5450⁹⁹⁶ - Change the print-bind output to be more precise
- PR #5449⁹⁹⁷ - Adapt swap_ranges to C++ 20
- PR #5446⁹⁹⁸ - Use more verbose names in sender algorithms
- PR #5443⁹⁹⁹ - Properly support ASAN with MSVC
- PR #5441¹⁰⁰⁰ - Adding reference counting to thread_data
- PR #5429¹⁰⁰¹ - Scheduler executor
- PR #5428¹⁰⁰² - Adapt datapar copy
- PR #5421¹⁰⁰³ - Update CI base image to use clang-format 11
- PR #5410¹⁰⁰⁴ - Add ranges starts_with and ends_with algorithms
- PR #5383¹⁰⁰⁵ - Tentatively remove runtime_registration_wrapper from cuda futures
- PR #5377¹⁰⁰⁶ - Fewer Asio includes and more precompiled headers
- PR #5329¹⁰⁰⁷ - Sender overloads for parallel algorithms
- PR #5313¹⁰⁰⁸ - Rearrange modules between libraries
- PR #5283¹⁰⁰⁹ - Require minimum C++17 and change CUDA handling
- PR #5241¹⁰¹⁰ - Adapt min_element, max_element and minmax_element to C++20

⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5461>

⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5460>

⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5459>

⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5458>

⁹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5455>

⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5454>

994 <https://github.com/STELLAR-GROUP/hpx/pull/5453>995 <https://github.com/STELLAR-GROUP/hpx/pull/5452>996 <https://github.com/STELLAR-GROUP/hpx/pull/5450>997 <https://github.com/STELLAR-GROUP/hpx/pull/5449>998 <https://github.com/STELLAR-GROUP/hpx/pull/5446>999 <https://github.com/STELLAR-GROUP/hpx/pull/5443>1000 <https://github.com/STELLAR-GROUP/hpx/pull/5441>1001 <https://github.com/STELLAR-GROUP/hpx/pull/5429>1002 <https://github.com/STELLAR-GROUP/hpx/pull/5428>1003 <https://github.com/STELLAR-GROUP/hpx/pull/5421>1004 <https://github.com/STELLAR-GROUP/hpx/pull/5410>1005 <https://github.com/STELLAR-GROUP/hpx/pull/5383>1006 <https://github.com/STELLAR-GROUP/hpx/pull/5377>1007 <https://github.com/STELLAR-GROUP/hpx/pull/5329>1008 <https://github.com/STELLAR-GROUP/hpx/pull/5313>1009 <https://github.com/STELLAR-GROUP/hpx/pull/5283>1010 <https://github.com/STELLAR-GROUP/hpx/pull/5241>

2.10.4 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 #5494¹⁰¹¹ - Rename *simdpar* execution policy to *par_simd*
- Issue #5488¹⁰¹² - *hpx::util::bind* doesn't bounds-check placeholders
- Issue #5486¹⁰¹³ - Possible V1.7.1 release

Closed pull requests

- PR #5500¹⁰¹⁴ - Minor bug fix in transform exclusive and inclusive scan tests
- PR #5499¹⁰¹⁵ - Rename *simdpar* to *par_simd*
- PR #5489¹⁰¹⁶ - Adding bound-checking for bind placeholders
- PR #5485¹⁰¹⁷ - Add Asio version check
- PR #5482¹⁰¹⁸ - Change extra archive data to rely on uniqueness of a variable address, not a function address
- PR #5448¹⁰¹⁹ - More fixes to enable for all types to be assumed to be bitwise copyable

¹⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/5494>

¹⁰¹² <https://github.com/STELLAR-GROUP/hpx/issues/5488>

¹⁰¹³ <https://github.com/STELLAR-GROUP/hpx/issues/5486>

¹⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5500>

¹⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5499>

¹⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5489>

¹⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5485>

¹⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5482>

¹⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5448>

- PR #5445¹⁰²⁰ - Improve performance of Spinlocks
- PR #5444¹⁰²¹ - Adapt transform_inclusive_scan to C++ 20
- PR #5440¹⁰²² - Adapt transform_exclusive_scan to C++ 20
- PR #5439¹⁰²³ - Adapt inclusive_scan to C++ 20
- PR #5436¹⁰²⁴ - Adapt exclusive_scan to C++20

2.10.5 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.

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`,

¹⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5445>

¹⁰²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5444>

¹⁰²² <https://github.com/STELLAR-GROUP/hpx/pull/5440>

¹⁰²³ <https://github.com/STELLAR-GROUP/hpx/pull/5439>

¹⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5436>

- 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_nonserv`. 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 #5423¹⁰²⁵ - Fix lvalue-ref qualified connect for `when_all-sender`
- Issue #5412¹⁰²⁶ - Link error
- Issue #5397¹⁰²⁷ - Performance regression in thread annotations
- Issue #5395¹⁰²⁸ - HPX 1.7.0-rc1 fails to build icw APEX + OTF2
- Issue #5385¹⁰²⁹ - HPX 1.7 crashes on Piz Daint > 64 nodes
- Issue #5380¹⁰³⁰ - CMake should search for asio package installed on the system
- Issue #5378¹⁰³¹ - HPX 1.7.0 stopped building on Fedora
- Issue #5369¹⁰³² - HPX 1.6 and master hangs on Summit for > 64 nodes
- Issue #5358¹⁰³³ - HPX init fails for single-core environments
- Issue #5345¹⁰³⁴ - Rename P2220 property CPOs?
- Issue #5333¹⁰³⁵ - HPX does not compile on the new Mac OSX using the M1 chip
- Issue #5317¹⁰³⁶ - Consider making hipblas optional
- Issue #5306¹⁰³⁷ - asio fails to build with CUDA 10.0
- Issue #5294¹⁰³⁸ - `execution::on` should be based on `execution::schedule`
- Issue #5275¹⁰³⁹ - HPX V1.6.0 fails on Fedora release
- Issue #5270¹⁰⁴⁰ - HPX-1.6.0 fails to build on Windows 10
- Issue #5257¹⁰⁴¹ - Allow triggering the output of OS thread affinity from configuration settings
- Issue #5246¹⁰⁴² - HPX fails to build on ppc64le
- Issue #5232¹⁰⁴³ - Annotation using `hpx::util::annotated_function` not working

¹⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5423>

¹⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5412>

¹⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5397>

¹⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5395>

¹⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5385>

¹⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5380>

¹⁰³¹ <https://github.com/STELLAR-GROUP/hpx/issues/5378>

¹⁰³² <https://github.com/STELLAR-GROUP/hpx/issues/5369>

¹⁰³³ <https://github.com/STELLAR-GROUP/hpx/issues/5358>

¹⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5345>

¹⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5333>

¹⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5317>

¹⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5306>

¹⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5294>

¹⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5275>

¹⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5270>

¹⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/5257>

¹⁰⁴² <https://github.com/STELLAR-GROUP/hpx/issues/5246>

¹⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/5232>

- Issue #5222¹⁰⁴⁴ - Build and link errors with ittnotify enabled
- Issue #5204¹⁰⁴⁵ - Move algorithms to tag_fallback_dispatch
- Issue #5163¹⁰⁴⁶ - Remove module-specific compatibility and deprecation options
- Issue #5161¹⁰⁴⁷ - Bump required CMake version to 3.17
- Issue #5143¹⁰⁴⁸ - Searching for HPX-Application to generate work on multiple Nodes

Closed pull requests

- PR #5438¹⁰⁴⁹ - Delete datapar/foreach_tests.hpp
- PR #5437¹⁰⁵⁰ - Add back explicit -pthread flags when available
- PR #5435¹⁰⁵¹ - This adds support for systems that assume all types are bitwise serializable by default
- PR #5434¹⁰⁵² - Update CUDA polling logging to be more verbose
- PR #5433¹⁰⁵³ - Fix when_all_sender connect for references
- PR #5432¹⁰⁵⁴ - Add deprecation warnings for v1.8
- PR #5431¹⁰⁵⁵ - Rename the new P0443/P2300 executor to thread_pool_scheduler
- PR #5430¹⁰⁵⁶ - Revert “Adding the missing defined for HPX_HAVE_DEPRECATED_WARNINGS”
- PR #5427¹⁰⁵⁷ - Removing unneeded typedef
- PR #5426¹⁰⁵⁸ - Adding more concept checks for sender/receiver algorithms
- PR #5425¹⁰⁵⁹ - Adding the missing defined for HPX_HAVE_DEPRECATED_WARNINGS
- PR #5424¹⁰⁶⁰ - Disable Vc in final docker image created in CI
- PR #5422¹⁰⁶¹ - Adding execution::experimental::bulk algorithm
- PR #5420¹⁰⁶² - Update logic to find threading library
- PR #5418¹⁰⁶³ - Reduce max size and number of files in ccache cache
- PR #5417¹⁰⁶⁴ - Final release notes for 1.7.0

¹⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5222>

¹⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5204>

¹⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5163>

¹⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5161>

¹⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5143>

¹⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5438>

¹⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5437>

¹⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/5435>

¹⁰⁵² <https://github.com/STELLAR-GROUP/hpx/pull/5434>

¹⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/5433>

¹⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5432>

¹⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5431>

¹⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5430>

¹⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5427>

¹⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5426>

¹⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5425>

¹⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5424>

¹⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5422>

¹⁰⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5420>

¹⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5418>

¹⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5417>

- PR #5416¹⁰⁶⁵ - Adapt `uninitialized_value_construct` and `uninitialized_value_construct_n` to C++ 20
- PR #5415¹⁰⁶⁶ - Adapt `uninitialized_default_construct` and `uninitialized_default_construct_n` to C++ 20
- PR #5414¹⁰⁶⁷ - Improve integration of futures and senders
- PR #5413¹⁰⁶⁸ - Fixing sender/receiver code base to compile with MSVC
- PR #5407¹⁰⁶⁹ - Handle exceptions thrown during initialization of parcel handler
- PR #5406¹⁰⁷⁰ - Simplify dispatching to annotation handlers
- PR #5405¹⁰⁷¹ - Fetch Asio automatically in perftests CI
- PR #5403¹⁰⁷² - Create generic executor that adds annotations to any other executor
- PR #5402¹⁰⁷³ - Adapt `uninitialized_fill` and `uninitialized_fill_n` to C++ 20
- PR #5401¹⁰⁷⁴ - Modernize a variety of facilities related to parallel algorithms
- PR #5400¹⁰⁷⁵ - Fix sliding semaphore test
- PR #5399¹⁰⁷⁶ - Rename leftover `tagFallbackInvoke` to `tagFallbackDispatch`
- PR #5398¹⁰⁷⁷ - Improve logging in AGAS symbol namespace
- PR #5396¹⁰⁷⁸ - Introduce compatibility layer for collective operations
- PR #5394¹⁰⁷⁹ - Enable OTF2 in APEX CI configuration
- PR #5393¹⁰⁸⁰ - Update APEX tag
- PR #5392¹⁰⁸¹ - Fixing wrong usage of `std::forward`
- PR #5391¹⁰⁸² - Fix forwarding in `transform_receiver` constructor
- PR #5390¹⁰⁸³ - Make sure shared priority scheduler steals tasks on the current NUMA domain when (core) stealing is enabled
- PR #5389¹⁰⁸⁴ - Adapt `uninitialized_move` and `uninitialized_move_n` to C++ 20
- PR #5388¹⁰⁸⁵ - Fixing `gather_there` for used with lvalue reference argument
- PR #5387¹⁰⁸⁶ - Extend thread state logging and change default stealing parameters

1065 <https://github.com/STELLAR-GROUP/hpx/pull/5416>

1066 <https://github.com/STELLAR-GROUP/hpx/pull/5415>

1067 <https://github.com/STELLAR-GROUP/hpx/pull/5414>

1068 <https://github.com/STELLAR-GROUP/hpx/pull/5413>

1069 <https://github.com/STELLAR-GROUP/hpx/pull/5407>

1070 <https://github.com/STELLAR-GROUP/hpx/pull/5406>

1071 <https://github.com/STELLAR-GROUP/hpx/pull/5405>

1072 <https://github.com/STELLAR-GROUP/hpx/pull/5403>

1073 <https://github.com/STELLAR-GROUP/hpx/pull/5402>

1074 <https://github.com/STELLAR-GROUP/hpx/pull/5401>

1075 <https://github.com/STELLAR-GROUP/hpx/pull/5400>

1076 <https://github.com/STELLAR-GROUP/hpx/pull/5399>

1077 <https://github.com/STELLAR-GROUP/hpx/pull/5398>

1078 <https://github.com/STELLAR-GROUP/hpx/pull/5396>

1079 <https://github.com/STELLAR-GROUP/hpx/pull/5394>

1080 <https://github.com/STELLAR-GROUP/hpx/pull/5393>

1081 <https://github.com/STELLAR-GROUP/hpx/pull/5392>

1082 <https://github.com/STELLAR-GROUP/hpx/pull/5391>

1083 <https://github.com/STELLAR-GROUP/hpx/pull/5390>

1084 <https://github.com/STELLAR-GROUP/hpx/pull/5389>

1085 <https://github.com/STELLAR-GROUP/hpx/pull/5388>

1086 <https://github.com/STELLAR-GROUP/hpx/pull/5387>

- PR #5386¹⁰⁸⁷ - Attempt to fix the startup hang with nodes > 32
- PR #5384¹⁰⁸⁸ - Remove HPX 1.5.0 deprecations
- PR #5382¹⁰⁸⁹ - Prefer installed Asio before considering FetchContent
- PR #5379¹⁰⁹⁰ - Allow using pre-downloaded (not installed) versions of Asio and/or Apex
- PR #5376¹⁰⁹¹ - Remove unnecessary explicit listing of library modules.rst files in CMakeLists.txt
- PR #5375¹⁰⁹² - Slight performance improvement for `hpx::copy` and `hpx::move` et.al.
- PR #5374¹⁰⁹³ - Remove unnecessary moves from future sender implementations
- PR #5373¹⁰⁹⁴ - More changes to clang-cuda Jenkins configuration
- PR #5372¹⁰⁹⁵ - Slight improvements to `min/max/minmax_element` algorithms
- PR #5371¹⁰⁹⁶ - Adapt `uninitialized_copy` and `uninitialized_copy_n` to C++ 20
- PR #5370¹⁰⁹⁷ - Decay types in `just_sender value_types` to match stored types
- PR #5367¹⁰⁹⁸ - Disable `pkgconfig` by default again on macOS
- PR #5365¹⁰⁹⁹ - Use `ccache` for Jenkins builds on Piz Daint
- PR #5363¹¹⁰⁰ - Update cudatoolkit module name in clang-cuda Jenkins configuration
- PR #5362¹¹⁰¹ - Adding `channel_communicator`
- PR #5361¹¹⁰² - Fix compilation with MPI enabled
- PR #5360¹¹⁰³ - Update APEX and asio tags
- PR #5359¹¹⁰⁴ - Fix check for pu-step in single-core case
- PR #5357¹¹⁰⁵ - Making sure collective operations can be reused by preallocating communicator
- PR #5356¹¹⁰⁶ - Update API documentation
- PR #5355¹¹⁰⁷ - Make the `sequenced_executor processing_units_count` member function const
- PR #5354¹¹⁰⁸ - Making sure `default_stack_size` is defined whenever declared
- PR #5353¹¹⁰⁹ - Add CUDA timestamp support to HPX Hardware Clock

¹⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5386>

¹⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5384>

¹⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5382>

¹⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5379>

¹⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5376>

¹⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5375>

¹⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5374>

¹⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5373>

¹⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5372>

¹⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5371>

¹⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5370>

¹⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5367>

¹⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5365>

¹¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5363>

¹¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5362>

¹¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5361>

¹¹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5360>

¹¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5359>

¹¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5357>

¹¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5356>

¹¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5355>

¹¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5354>

¹¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5353>

- PR #5352¹¹¹⁰ - Adding missing includes
- PR #5351¹¹¹¹ - Adding enable_logging/disable_logging API functions
- PR #5350¹¹¹² - Adapt lexicographical_compare to C++20
- PR #5349¹¹¹³ - Update minimum boost version needed on the docs
- PR #5348¹¹¹⁴ - Rename tag_invoke and related facilities to tag_dispatch
- PR #5347¹¹¹⁵ - Remove make_ prefix for executor properties
- PR #5346¹¹¹⁶ - Remove and disable compatibility options for 1.7.0
- PR #5343¹¹¹⁷ - Fix timed_executor static cast conversion
- PR #5342¹¹¹⁸ - Refactor CUDA event polling
- PR #5341¹¹¹⁹ - Adding make_with_annotation and get_annotation properties
- PR #5339¹¹²⁰ - Making sure hpx::util::hardware::timestamp() is always defined
- PR #5338¹¹²¹ - Fixing timed_executor specializations of customization points
- PR #5335¹¹²² - Make partial_algorithm work with any number of arguments
- PR #5334¹¹²³ - Follow up iter_sent include on #5225
- PR #5332¹¹²⁴ - Simplify tag_invoke and friends
- PR #5331¹¹²⁵ - More work on cleaning up executor CPOs
- PR #5330¹¹²⁶ - Add option to disable pkgconfig generation
- PR #5328¹¹²⁷ - Adapt data parallel support using std-simd
- PR #5327¹¹²⁸ - Fix missing ifdef HPX_SMT_PAUSE
- PR #5326¹¹²⁹ - Adding resize() to serialize_buffer allowing to shrink its size
- PR #5324¹¹³⁰ - Add get member functions to async_rw_mutex proxy objects for explicitly getting the wrapped value
- PR #5323¹¹³¹ - Add keep_future algorithm
- PR #5322¹¹³² - Replace executor customization point implementations with tag_invoke

1110 <https://github.com/STELLAR-GROUP/hpx/pull/5352>

1111 <https://github.com/STELLAR-GROUP/hpx/pull/5351>

1112 <https://github.com/STELLAR-GROUP/hpx/pull/5350>

1113 <https://github.com/STELLAR-GROUP/hpx/pull/5349>

1114 <https://github.com/STELLAR-GROUP/hpx/pull/5348>

1115 <https://github.com/STELLAR-GROUP/hpx/pull/5347>

1116 <https://github.com/STELLAR-GROUP/hpx/pull/5346>

1117 <https://github.com/STELLAR-GROUP/hpx/pull/5343>

1118 <https://github.com/STELLAR-GROUP/hpx/pull/5342>

1119 <https://github.com/STELLAR-GROUP/hpx/pull/5341>

1120 <https://github.com/STELLAR-GROUP/hpx/pull/5339>

1121 <https://github.com/STELLAR-GROUP/hpx/pull/5338>

1122 <https://github.com/STELLAR-GROUP/hpx/pull/5335>

1123 <https://github.com/STELLAR-GROUP/hpx/pull/5334>

1124 <https://github.com/STELLAR-GROUP/hpx/pull/5332>

1125 <https://github.com/STELLAR-GROUP/hpx/pull/5331>

1126 <https://github.com/STELLAR-GROUP/hpx/pull/5330>

1127 <https://github.com/STELLAR-GROUP/hpx/pull/5328>

1128 <https://github.com/STELLAR-GROUP/hpx/pull/5327>

1129 <https://github.com/STELLAR-GROUP/hpx/pull/5326>

1130 <https://github.com/STELLAR-GROUP/hpx/pull/5324>

1131 <https://github.com/STELLAR-GROUP/hpx/pull/5323>

1132 <https://github.com/STELLAR-GROUP/hpx/pull/5322>

- PR #5321¹¹³³ - Separate segmented algorithms for reduce
- PR #5320¹¹³⁴ - Fix `is_sender` trait and other small fixes to p0443 traits
- PR #5319¹¹³⁵ - gcc 11.1 c++20 build fixes
- PR #5318¹¹³⁶ - Make hipblas dependency optional as not always available
- PR #5316¹¹³⁷ - Attempt to fix checking for libatomic
- PR #5315¹¹³⁸ - Add explicit keyword to fixture constructor
- PR #5314¹¹³⁹ - Fix a race condition in async mpi affecting limiting executor
- PR #5312¹¹⁴⁰ - Use local runtime and local headers in local-only modules and tests
- PR #5311¹¹⁴¹ - Add GCC 11 builder to jenkins
- PR #5310¹¹⁴² - Adding `hpx::execution::experimental::task_group`
- PR #5309¹¹⁴³ - Separate datapar
- PR #5308¹¹⁴⁴ - Separate segmented algorithms for `find`, `find_if`, `find_if_not`
- PR #5307¹¹⁴⁵ - Separate segmented algorithms for `fill` and `generate`
- PR #5304¹¹⁴⁶ - Fix compilation of sender CPOs with nvcc
- PR #5300¹¹⁴⁷ - Remove PRIVATE flag that was propagated into the LANGUAGES
- PR #5298¹¹⁴⁸ - Separate datapar
- PR #5297¹¹⁴⁹ - Specify exact cmake and ninja versions when loading them in jenkins jobs
- PR #5295¹¹⁵⁰ - Update clang-newest configuration to use clang 12 and Boost 1.76.0
- PR #5293¹¹⁵¹ - Fix Clang 11 cuda_future test bug
- PR #5292¹¹⁵² - Add `async_rw_mutex` based on senders
- PR #5291¹¹⁵³ - “Fix” termination detection
- PR #5290¹¹⁵⁴ - Fixed source file line statements in examples documentation
- PR #5289¹¹⁵⁵ - Allow splitting of futures holding `std::tuple`

¹¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/5321>¹¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5320>¹¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5319>¹¹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5318>¹¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5316>¹¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5315>¹¹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5314>¹¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5312>¹¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5311>¹¹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5310>¹¹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5309>¹¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5308>¹¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5307>¹¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5304>¹¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5300>¹¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5298>¹¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5297>¹¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5295>¹¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/5293>¹¹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/5292>¹¹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/5291>¹¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5290>¹¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5289>

- PR #5288¹¹⁵⁶ - Move algorithms to `tag_fallback_invoke`
- PR #5287¹¹⁵⁷ - Move algorithms to `tag_fallback_invoke`
- PR #5285¹¹⁵⁸ - Fix clang-format failure on master
- PR #5284¹¹⁵⁹ - Replacing `util::function_nonser` on `std::function` in `hpx_init`
- PR #5282¹¹⁶⁰ - Update Boost for daint 20.11 after update
- PR #5281¹¹⁶¹ - Fix Segmentation fault on `foreach_datapar_zipiter`
- PR #5280¹¹⁶² - Avoid modulo by zero in `counting_iterator` test
- PR #5279¹¹⁶³ - Fix more GCC 10 deprecation warnings
- PR #5277¹¹⁶⁴ - Small fixes and improvements to CUDA/MPI polling
- PR #5276¹¹⁶⁵ - Fix typo in docs
- PR #5274¹¹⁶⁶ - More P1897 algorithms
- PR #5273¹¹⁶⁷ - Retry CDash submissions on failure
- PR #5272¹¹⁶⁸ - Fix bogus deprecation warnings with GCC 10
- PR #5271¹¹⁶⁹ - Correcting target ids for `symbol_namespace::iterate`
- PR #5268¹¹⁷⁰ - Adding generic `require`, `require_concept`, and `query` properties
- PR #5267¹¹⁷¹ - Support annotations in `hpx::transform_reduce`
- PR #5266¹¹⁷² - Making late command line options available for local runtime
- PR #5265¹¹⁷³ - Leverage `no_unique_address` for `member_pack`
- PR #5264¹¹⁷⁴ - Adopt format in more places
- PR #5262¹¹⁷⁵ - Install HPX in Rostam Jenkins jobs
- PR #5261¹¹⁷⁶ - Limit Rostam Jenkins jobs to marvin partition temporarily
- PR #5260¹¹⁷⁷ - Separate segmented algorithms for `transform_reduce`
- PR #5259¹¹⁷⁸ - Making sure late command line options are recognized as configuration options

¹¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5288>

¹¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5287>

¹¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5285>

¹¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5284>

¹¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5282>

¹¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5281>

¹¹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5280>

¹¹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5279>

¹¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5277>

¹¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5276>

¹¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5274>

¹¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5273>

¹¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5272>

¹¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5271>

¹¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5268>

¹¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/5267>

¹¹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/5266>

¹¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/5265>

1174 <https://github.com/STELLAR-GROUP/hpx/pull/5264>1175 <https://github.com/STELLAR-GROUP/hpx/pull/5262>1176 <https://github.com/STELLAR-GROUP/hpx/pull/5261>1177 <https://github.com/STELLAR-GROUP/hpx/pull/5260>1178 <https://github.com/STELLAR-GROUP/hpx/pull/5259>

- PR #5258¹¹⁷⁹ - Allow for HPX algorithms being invoked with std execution policies
- PR #5256¹¹⁸⁰ - Separate segmented algorithms for transform
- PR #5255¹¹⁸¹ - Future/sender adapters
- PR #5254¹¹⁸² - Fixing datapar
- PR #5253¹¹⁸³ - Add utility to format ranges
- PR #5252¹¹⁸⁴ - Remove uses of Boost.Bimap
- PR #5251¹¹⁸⁵ - Banish <iostream> from library headers
- PR #5250¹¹⁸⁶ - Try fixing vc circle ci
- PR #5249¹¹⁸⁷ - Adding missing header
- PR #5248¹¹⁸⁸ - Use old Piz Daint modules after upgrade
- PR #5247¹¹⁸⁹ - Significantly speedup simple `for_each`, `for_loop`, and `transform`
- PR #5245¹¹⁹⁰ - P1897 `operator|` overloads
- PR #5244¹¹⁹¹ - P1897 `when_all`
- PR #5243¹¹⁹² - Make sure `HPX_DEBUG` is set based on HPX's build type, not consuming project's build type
- PR #5242¹¹⁹³ - Moving last files unrelated to parcel layer to modules
- PR #5240¹¹⁹⁴ - change namespace for `transform_loop.hpp`
- PR #5238¹¹⁹⁵ - Make sure annotations are used in the binary transform
- PR #5237¹¹⁹⁶ - Add P1897 `just`, `just_on`, and `on` algorithms
- PR #5236¹¹⁹⁷ - Add an example demonstrating the use of the `invoke_function_action` facility
- PR #5235¹¹⁹⁸ - Attempting to fix datapar compilation issues
- PR #5234¹¹⁹⁹ - Fix small typo in `--hpx:local` option description
- PR #5233¹²⁰⁰ - Only find Boost.Iostreams if required for plugins
- PR #5231¹²⁰¹ - Sort printed config options

¹¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5258>

¹¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5256>

¹¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5255>

¹¹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5254>

¹¹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5253>

¹¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5252>

1185 <https://github.com/STELLAR-GROUP/hpx/pull/5251>1186 <https://github.com/STELLAR-GROUP/hpx/pull/5250>1187 <https://github.com/STELLAR-GROUP/hpx/pull/5249>1188 <https://github.com/STELLAR-GROUP/hpx/pull/5248>1189 <https://github.com/STELLAR-GROUP/hpx/pull/5247>1190 <https://github.com/STELLAR-GROUP/hpx/pull/5245>1191 <https://github.com/STELLAR-GROUP/hpx/pull/5244>1192 <https://github.com/STELLAR-GROUP/hpx/pull/5243>1193 <https://github.com/STELLAR-GROUP/hpx/pull/5242>1194 <https://github.com/STELLAR-GROUP/hpx/pull/5240>1195 <https://github.com/STELLAR-GROUP/hpx/pull/5238>1196 <https://github.com/STELLAR-GROUP/hpx/pull/5237>1197 <https://github.com/STELLAR-GROUP/hpx/pull/5236>1198 <https://github.com/STELLAR-GROUP/hpx/pull/5235>1199 <https://github.com/STELLAR-GROUP/hpx/pull/5234>1200 <https://github.com/STELLAR-GROUP/hpx/pull/5233>1201 <https://github.com/STELLAR-GROUP/hpx/pull/5231>

- PR #5230¹²⁰² - Fix C++20 replace algo adaptation misses
- PR #5229¹²⁰³ - Remove leftover Boost include from sync_wait.hpp
- PR #5228¹²⁰⁴ - Print module name only if it has custom configuration settings
- PR #5227¹²⁰⁵ - Update .codespell_whitelist
- PR #5226¹²⁰⁶ - Use new docker image in all CircleCI steps
- PR #5225¹²⁰⁷ - Adapt reverse to C++20
- PR #5224¹²⁰⁸ - Separate segmented algorithms for none_of, any_of and all_of
- PR #5223¹²⁰⁹ - Fixing build system for ittnotify
- PR #5221¹²¹⁰ - Moving LCO related files to modules
- PR #5220¹²¹¹ - Separate segmented algorithms for count and count_if
- PR #5218¹²¹² - Separate segmented algorithms for adjacent_find
- PR #5217¹²¹³ - Add a HIP github action
- PR #5215¹²¹⁴ - Update ROCm to 4.0.1 on Rostam
- PR #5214¹²¹⁵ - Fix clang-format error in sender.hpp
- PR #5213¹²¹⁶ - Removing ESSENTIAL option to the doc example
- PR #5212¹²¹⁷ - Separate segmented algorithms for for_each_n
- PR #5211¹²¹⁸ - Minor adapted algos fixes
- PR #5210¹²¹⁹ - Fixing is_invocable deprecation warnings
- PR #5209¹²²⁰ - Moving more files into modules (actions, components, init_runtime, etc.)
- PR #5208¹²²¹ - Add examples and explanation on when tag_fallback/priority are useful
- PR #5207¹²²² - Always define HPX_COMPUTE_HOST_CODE for host code
- PR #5206¹²²³ - Add formatting exceptions for libhpx to create_module_skeleton.py
- PR #5205¹²²⁴ - Moving all distribution policies into modules

¹²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5230>

¹²⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5229>

¹²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5228>

¹²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5227>

¹²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5226>

¹²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5225>

¹²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5224>

¹²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5223>

¹²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5221>

¹²¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5220>

¹²¹² <https://github.com/STELLAR-GROUP/hpx/pull/5218>

¹²¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5217>

¹²¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5215>

¹²¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5214>

¹²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5213>

¹²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5212>

¹²¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5211>

¹²¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5210>

¹²²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5209>

¹²²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5208>

¹²²² <https://github.com/STELLAR-GROUP/hpx/pull/5207>

¹²²³ <https://github.com/STELLAR-GROUP/hpx/pull/5206>

¹²²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5205>

- PR #5203¹²²⁵ - Move copy algorithms to `tag_fallback_invoke`
- PR #5202¹²²⁶ - Make `HPX_WITH_PSEUDO_DEPENDENCIES` a cache variable
- PR #5201¹²²⁷ - Replaced `tag_invoke` with `tag_fallback_invoke` for `adjacent_find` algorithm
- PR #5200¹²²⁸ - Moving files to (distributed) runtime module
- PR #5199¹²²⁹ - Update ICC module name on Piz Daint Jenkins configuration
- PR #5198¹²³⁰ - Add doxygen documentation for `thread_schedule_hint`
- PR #5197¹²³¹ - Attempt to fix compilation of context implementations with unity build enabled
- PR #5196¹²³² - Re-enable component tests
- PR #5195¹²³³ - Moving files related to colocation logic
- PR #5194¹²³⁴ - Another attempt at fixing the Fedora 35 problem
- PR #5193¹²³⁵ - Components module
- PR #5192¹²³⁶ - Adapt `replace(_if)` to C++20
- PR #5190¹²³⁷ - Set compatibility headers by default to on
- PR #5188¹²³⁸ - Bump Boost minimum version to 1.71.0
- PR #5187¹²³⁹ - Force CMake to set the `-std=c++XX` flag
- PR #5186¹²⁴⁰ - Remove message to print .cu extension whenever .cu files are encountered
- PR #5185¹²⁴¹ - Remove some minor unnecessary CMake options
- PR #5184¹²⁴² - Remove some leftover `HPX_WITH_*_SCHEDULER` uses
- PR #5183¹²⁴³ - Remove dependency on boost/iterators/iterator_categories.hpp
- PR #5182¹²⁴⁴ - Fixing Fedora 35 for Power architectures
- PR #5181¹²⁴⁵ - Bump version number and tag post 1.6.0 release
- PR #5180¹²⁴⁶ - Fix htts_v2 tests linking
- PR #5179¹²⁴⁷ - Make sure `--hpx:local` command line option is respected with networking is off but distributed runtime is on

¹²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5203>

¹²²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5202>

¹²²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5201>

¹²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5200>

¹²²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5199>

¹²³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5198>

¹²³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5197>

¹²³² <https://github.com/STELLAR-GROUP/hpx/pull/5196>

¹²³³ <https://github.com/STELLAR-GROUP/hpx/pull/5195>

¹²³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5194>

¹²³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5193>

¹²³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5192>

¹²³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5190>

¹²³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5188>

¹²³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5187>

¹²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5186>

¹²⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5185>

¹²⁴² <https://github.com/STELLAR-GROUP/hpx/pull/5184>

¹²⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/5183>

¹²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5182>

¹²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5181>

¹²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5180>

¹²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5179>

- PR #5177¹²⁴⁸ - Remove module cmake options
- PR #5176¹²⁴⁹ - Starting to separate segmented algorithms: `for_each`
- PR #5174¹²⁵⁰ - Don't run segmented algorithms twice on CircleCI
- PR #5173¹²⁵¹ - Fetching APEX using cmake FetchContent
- PR #5172¹²⁵² - Add separate local-only entry point
- PR #5171¹²⁵³ - Remove `HPX_WITH_THREAD_SCHEDULERS` CMake option
- PR #5170¹²⁵⁴ - Add `HPX_WITH_PRECOMPILED_HEADERS` option
- PR #5166¹²⁵⁵ - Moving some action tests to modules
- PR #5165¹²⁵⁶ - Require cmake 3.17
- PR #5164¹²⁵⁷ - Move `thread_pool_suspension_helper` files to small utility module
- PR #5160¹²⁵⁸ - Adding checks ensuring modules are not cross-referenced from other module categories
- PR #5158¹²⁵⁹ - Replace boost::asio with standalone asio
- PR #5155¹²⁶⁰ - Allow logging when distributed runtime is off
- PR #5153¹²⁶¹ - Components module
- PR #5152¹²⁶² - Move more files to performance counter module
- PR #5150¹²⁶³ - Adapt `remove_copy(_if)` to C++20
- PR #5144¹²⁶⁴ - AGAS module
- PR #5125¹²⁶⁵ - Adapt `remove` and `remove_if` to C++20
- PR #5117¹²⁶⁶ - Attempt to fix segfaults assumed to be caused by `future_data` instances going out of scope.
- PR #5099¹²⁶⁷ - Allow mixing debug and release builds
- PR #5092¹²⁶⁸ - Replace spirit.qi with x3
- PR #5053¹²⁶⁹ - Add P0443r14 executor and a few P1897 algorithms
- PR #5044¹²⁷⁰ - Add performance test in jenkins and reports

¹²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5177>

¹²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5176>

¹²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5174>

¹²⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/5173>

¹²⁵² <https://github.com/STELLAR-GROUP/hpx/pull/5172>

¹²⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/5171>

¹²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5170>

¹²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5166>

¹²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5165>

¹²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5164>

¹²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5160>

¹²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5158>

¹²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5155>

¹²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5153>

1262 <https://github.com/STELLAR-GROUP/hpx/pull/5152>1263 <https://github.com/STELLAR-GROUP/hpx/pull/5150>1264 <https://github.com/STELLAR-GROUP/hpx/pull/5144>1265 <https://github.com/STELLAR-GROUP/hpx/pull/5125>1266 <https://github.com/STELLAR-GROUP/hpx/pull/5117>1267 <https://github.com/STELLAR-GROUP/hpx/pull/5099>1268 <https://github.com/STELLAR-GROUP/hpx/pull/5092>1269 <https://github.com/STELLAR-GROUP/hpx/pull/5053>1270 <https://github.com/STELLAR-GROUP/hpx/pull/5044>

2.10.6 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 P2220^{[1271](#)}. 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`, `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

¹²⁷¹ <https://wg21.link/p2220>

deprecated unscoped enumerations are disabled by `HPX_WITH_UNSCOPED_ENUM_COMPATIBILITY`. Newly deprecated unscoped enumerations have been given deprecation warnings and replaced by scoped enumerations. `hpx::promise` has been replaced with `hpx::distributed::promise`. `hpx::program_options` is a drop-in replacement for `boost::program_options`. `hpx::execution::parallel_executor` now has constructors which take a thread pool, covering the use case of `hpx::threads::executors::pool_executor`. A pool can be supplied with `hpx::resource::get_thread_pool`.

Closed issues

- Issue #5148¹²⁷² - `runtime_support.hpp` does not work with newer clang
- Issue #5147¹²⁷³ - Wrong results with parallel reduce
- Issue #5129¹²⁷⁴ - Missing specialization for `std::hash<hpx::thread::id>`
- Issue #5126¹²⁷⁵ - Use `std::string` for task annotations
- Issue #5115¹²⁷⁶ - Don't expect hwloc to always report Cores
- Issue #5113¹²⁷⁷ - Handle threadmanager exceptions during startup
- Issue #5112¹²⁷⁸ - libatomic problems causing unexpected fails
- Issue #5089¹²⁷⁹ - Remove non-BSL files
- Issue #5088¹²⁸⁰ - Unwrapping problem
- Issue #5087¹²⁸¹ - Remove hpxMP support
- Issue #5077¹²⁸² - PAPI counters are not accessible when HPX is installed
- Issue #5075¹²⁸³ - Make the structs in all `iter_sent.hpp` lower case
- Issue #5067¹²⁸⁴ - Bug `string_util/split.hpp`
- Issue #5049¹²⁸⁵ - Change back the hipcc jenkins config to the fury partition on rostam
- Issue #5038¹²⁸⁶ - Not all examples link in the latest HPX master
- Issue #5035¹²⁸⁷ - Build with `HPX_WITH_EXAMPLES` fails
- Issue #5019¹²⁸⁸ - Broken help string for hpx
- Issue #5016¹²⁸⁹ - `hpx::parallel::fill` fails compiling
- Issue #5014¹²⁹⁰ - Rename all .cc to .cpp and .hh to .hpp

¹²⁷² <https://github.com/STELLAR-GROUP/hpx/issues/5148>

¹²⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/5147>

¹²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5129>

¹²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5126>

¹²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5115>

¹²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5113>

¹²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5112>

¹²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5089>

¹²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5088>

¹²⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/5087>

¹²⁸² <https://github.com/STELLAR-GROUP/hpx/issues/5077>

¹²⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/5075>

¹²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/5067>

¹²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/5049>

¹²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/5038>

¹²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/5035>

¹²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/5019>

¹²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/5016>

¹²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/5014>

- Issue #4988¹²⁹¹ - MPI is not finalized if running with only one locality
- Issue #4978¹²⁹² - Change feature test macros to expand to zero/one
- Issue #4949¹²⁹³ - Crash when not enabling TCP parcelport
- Issue #4933¹²⁹⁴ - Improve test coverage for unused variable warnings etc.
- Issue #4878¹²⁹⁵ - HPX mpi async might call MPI_FINALIZE before app calls it
- Issue #4127¹²⁹⁶ - Local runtime entry-points

Closed pull requests

- PR #5178¹²⁹⁷ - Fix parallel remove/remove_copy/transform namespace references in docs
- PR #5169¹²⁹⁸ - Attempt to get Piz Daint jenkins setup running after maintenance
- PR #5168¹²⁹⁹ - Remove include of itself
- PR #5167¹³⁰⁰ - Fixing deprecation warnings that slipped through the net
- PR #5159¹³⁰¹ - Update APEX tag to 2.3.1
- PR #5154¹³⁰² - Splitting unit tests on circleci to avoid timeouts
- PR #5151¹³⁰³ - Use C++20 on clang-newest Jenkins CI configuration
- PR #5149¹³⁰⁴ - Rename 'module' symbols to avoid keyword conflict
- PR #5145¹³⁰⁵ - Adjust handling of CUDA/HIP options in CMake
- PR #5142¹³⁰⁶ - Store annotated_function annotations as std::strings
- PR #5140¹³⁰⁷ - Scheduler mode
- PR #5139¹³⁰⁸ - Fix path problem in pre-commit hook, add summary commit line
- PR #5138¹³⁰⁹ - Add program options variable map to resource partitioner init
- PR #5137¹³¹⁰ - Remove the use of boost::throw_exception
- PR #5136¹³¹¹ - Make sure codespell checks run on CircleCI
- PR #5132¹³¹² - Fixing spelling errors

¹²⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/4988>

¹²⁹² <https://github.com/STELLAR-GROUP/hpx/issues/4978>

¹²⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/4949>

¹²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4933>

¹²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4878>

¹²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4127>

¹²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5178>

¹²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5169>

¹²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5168>

¹³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5167>

¹³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5159>

¹³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5154>

¹³⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5151>

¹³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5149>

¹³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5145>

¹³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5142>

¹³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5140>

¹³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5139>

¹³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5138>

¹³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5137>

¹³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5136>

¹³¹² <https://github.com/STELLAR-GROUP/hpx/pull/5132>

- PR #5131¹³¹³ - Mark counting_iterator member functions as HPX_HOST_DEVICE
- PR #5130¹³¹⁴ - Adding specialization for std::hash<hpx::thread::id>
- PR #5128¹³¹⁵ - Fixing environment handling for FreeBSD
- PR #5127¹³¹⁶ - Fix typo in fibonacci documentation
- PR #5123¹³¹⁷ - Reduce vector sizes in partial sort benchmarks when running in debug mode
- PR #5122¹³¹⁸ - Making sure exceptions during runtime initialization are correctly reported
- PR #5121¹³¹⁹ - Working around hwloc limitation on certain platforms
- PR #5120¹³²⁰ - Fixing compatibility warnings in hpx::transform implementation
- PR #5119¹³²¹ - Use sequential_find and friends from separate detail header
- PR #5116¹³²² - Fix compilation with timer pool off
- PR #5114¹³²³ - Fix 5112 - make sure libatomic is used when needed
- PR #5109¹³²⁴ - Remove default runtime mode argument from init overload, again
- PR #5108¹³²⁵ - Refactor iter_sent.hpp to make structs lowercase
- PR #5107¹³²⁶ - Relax dataflow internals
- PR #5106¹³²⁷ - Change initialization of property CPOs to satisfy older nvcc versions
- PR #5104¹³²⁸ - Fix regeneration of two files that trigger unnecessary rebuilds
- PR #5103¹³²⁹ - Remove default runtime mode argument from start/init overloads
- PR #5102¹³³⁰ - Untie deprecated thread enums from the CMake option
- PR #5101¹³³¹ - Update APEX tag for 1.6.0
- PR #5100¹³³² - Bump minimum required Boost version to 1.66 and update CI configurations
- PR #5098¹³³³ - Minor fixes to public API listing
- PR #5097¹³³⁴ - Remove hpxMP support
- PR #5096¹³³⁵ - Remove fractals examples

¹³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/5131>

¹³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5130>

¹³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5128>

¹³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5127>

¹³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5123>

¹³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5122>

¹³¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5121>

¹³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5120>

¹³²¹ <https://github.com/STELLAR-GROUP/hpx/pull/5119>

¹³²² <https://github.com/STELLAR-GROUP/hpx/pull/5116>

¹³²³ <https://github.com/STELLAR-GROUP/hpx/pull/5114>

¹³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5109>

¹³²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5108>

¹³²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5107>

¹³²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5106>

¹³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5104>

¹³²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5103>

¹³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5102>

¹³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/5101>

¹³³² <https://github.com/STELLAR-GROUP/hpx/pull/5100>

¹³³³ <https://github.com/STELLAR-GROUP/hpx/pull/5098>

¹³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5097>

¹³³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5096>

- PR #5095¹³³⁶ - Use all AMD nodes again on rostam
- PR #5094¹³³⁷ - Attempt to remove macOS workaround for GH actions environment
- PR #5093¹³³⁸ - Remove verbs parcelport
- PR #5091¹³³⁹ - Avoid moving from lvalues
- PR #5090¹³⁴⁰ - Adopt C++20 std::endian
- PR #5085¹³⁴¹ - Update daint CI to use Boost 1.75.0
- PR #5084¹³⁴² - Disable compatibility options for 1.6.0 release
- PR #5083¹³⁴³ - Remove duplicated call to the `limiting_executor` in `future_overhead` test
- PR #5079¹³⁴⁴ - Add checks to make sure that MPI/CUDA polling is enabled/not disabled too early
- PR #5078¹³⁴⁵ - Add install lib directory to list of component search paths
- PR #5076¹³⁴⁶ - Fix a typo in the jenkins clang-newest cmake config
- PR #5074¹³⁴⁷ - Fixing warnings generated by MSVC
- PR #5073¹³⁴⁸ - Allow using noncopyable types with unwrapping
- PR #5072¹³⁴⁹ - Fix `is_convertible` args in `result_types`
- PR #5071¹³⁵⁰ - Fix unused parameters
- PR #5070¹³⁵¹ - Fix unused variables warnings in hipcc
- PR #5069¹³⁵² - Add support for sentinels to `adjacent_find`
- PR #5068¹³⁵³ - Fix string split function
- PR #5066¹³⁵⁴ - Adapt `search` to C++20 and Range TS
- PR #5065¹³⁵⁵ - Fix `hpx::range::adjacent_find` doxygen function signatures
- PR #5064¹³⁵⁶ - Refactor runtime configuration, command line handling, and resource partitioner
- PR #5063¹³⁵⁷ - Limit the device code guards to the distributed parts of the `future_overhead` bench
- PR #5061¹³⁵⁸ - Remove hipcc guards in examples and tests

¹³³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5095>

¹³³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5094>

¹³³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5093>

¹³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5091>

¹³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5090>

¹³⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/5085>

1342 <https://github.com/STELLAR-GROUP/hpx/pull/5084>1343 <https://github.com/STELLAR-GROUP/hpx/pull/5083>1344 <https://github.com/STELLAR-GROUP/hpx/pull/5079>1345 <https://github.com/STELLAR-GROUP/hpx/pull/5078>1346 <https://github.com/STELLAR-GROUP/hpx/pull/5076>1347 <https://github.com/STELLAR-GROUP/hpx/pull/5074>1348 <https://github.com/STELLAR-GROUP/hpx/pull/5073>1349 <https://github.com/STELLAR-GROUP/hpx/pull/5072>1350 <https://github.com/STELLAR-GROUP/hpx/pull/5071>1351 <https://github.com/STELLAR-GROUP/hpx/pull/5070>1352 <https://github.com/STELLAR-GROUP/hpx/pull/5069>1353 <https://github.com/STELLAR-GROUP/hpx/pull/5068>1354 <https://github.com/STELLAR-GROUP/hpx/pull/5066>1355 <https://github.com/STELLAR-GROUP/hpx/pull/5065>1356 <https://github.com/STELLAR-GROUP/hpx/pull/5064>1357 <https://github.com/STELLAR-GROUP/hpx/pull/5063>1358 <https://github.com/STELLAR-GROUP/hpx/pull/5061>

- PR #5060¹³⁵⁹ - Fix deprecation warnings generated by msvc
- PR #5059¹³⁶⁰ - Add warning about suspending/resuming the runtime in multi-locality scenarios
- PR #5057¹³⁶¹ - Fix unused variable warnings
- PR #5056¹³⁶² - Fix `hpx::util::get`
- PR #5055¹³⁶³ - Remove hipcc guards
- PR #5054¹³⁶⁴ - Fix typo
- PR #5051¹³⁶⁵ - Adapt transform to C++20
- PR #5050¹³⁶⁶ - Replace old init overloads in tests and examples
- PR #5048¹³⁶⁷ - Limit jenkins hipcc to the reno node
- PR #5047¹³⁶⁸ - Limit cuda jenkins run to nodes with exclusively Nvidia GPUs
- PR #5046¹³⁶⁹ - Convert thread and future enums to class enums
- PR #5043¹³⁷⁰ - Improve `hpxrun.py` for Phylanx
- PR #5042¹³⁷¹ - Add missing header to partial sort test
- PR #5041¹³⁷² - Adding Francisco Tapia's implementation of `partial_sort`
- PR #5040¹³⁷³ - Remove generated headers left behind from a previous configuration
- PR #5039¹³⁷⁴ - Fix GCC 10 release builds
- PR #5037¹³⁷⁵ - Add `is_invocable` typedefs to top-level hpx namespace and public API list
- PR #5036¹³⁷⁶ - Deprecate `hpx::util::decay` in favor of `std::decay`
- PR #5034¹³⁷⁷ - Use versioned container image on CircleCI
- PR #5033¹³⁷⁸ - Implement P2220 properties module
- PR #5032¹³⁷⁹ - Do codespell comparison only on files changed from common ancestor
- PR #5031¹³⁸⁰ - Moving traits files to `actions_base`
- PR #5030¹³⁸¹ - Add codespell version print in circleci

¹³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5060>

¹³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5059>

¹³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/5057>

¹³⁶² <https://github.com/STELLAR-GROUP/hpx/pull/5056>

¹³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/5055>

¹³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5054>

¹³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5051>

¹³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5050>

¹³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5048>

¹³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5047>

¹³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5046>

¹³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5043>

¹³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/5042>

¹³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/5041>

¹³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/5040>

¹³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5039>

¹³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5037>

¹³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5036>

¹³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5034>

¹³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5033>

¹³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5032>

¹³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5031>

¹³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/5030>

- PR #5029¹³⁸² - Work around problems in GitHub actions macOS builder
- PR #5028¹³⁸³ - Moving move files to naming and naming_base
- PR #5027¹³⁸⁴ - Lessen constraints on certain algorithm arguments
- PR #5025¹³⁸⁵ - Adapt is_sorted and is_sorted_until to C++20
- PR #5024¹³⁸⁶ - Moving naming_base to full modules
- PR #5022¹³⁸⁷ - Remove C language from CMakeLists.txt
- PR #5021¹³⁸⁸ - Warn about unused arguments given to add_hpx_module
- PR #5020¹³⁸⁹ - Fixing help string
- PR #5018¹³⁹⁰ - Update CSCS jenkins configuration to clang 11
- PR #5017¹³⁹¹ - Fixing broken backwards compatibility for hpx::parallel::fill
- PR #5015¹³⁹² - Detect if generated global header conflicts with explicitly listed module headers
- PR #5012¹³⁹³ - Properly reset pointer tracking data in output_archive
- PR #5011¹³⁹⁴ - Inspect command line tweaks
- PR #5010¹³⁹⁵ - Creating AGAS module
- PR #5009¹³⁹⁶ - Replace boost::system::error_code with std::error_code
- PR #5008¹³⁹⁷ - Replace uses of boost::detail::spinlock
- PR #5007¹³⁹⁸ - Bump minimal Boost version to 1.65.0
- PR #5006¹³⁹⁹ - Adapt is_partitioned to C++20
- PR #5005¹⁴⁰⁰ - Making sure reduce_by_key compiles again
- PR #5004¹⁴⁰¹ - Fixing template specializations that make extra archive data types unique across module boundaries
- PR #5003¹⁴⁰² - Relax dataflow argument constraints
- PR #5001¹⁴⁰³ - Add <random> inspect check
- PR #4999¹⁴⁰⁴ - Attempt to fix MacOS Github action error

¹³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/5029>

¹³⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/5028>

¹³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5027>

¹³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5025>

¹³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5024>

¹³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5022>

¹³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5021>

¹³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5020>

¹³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5018>

¹³⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/5017>

¹³⁹² <https://github.com/STELLAR-GROUP/hpx/pull/5015>

¹³⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/5012>

¹³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/5011>

¹³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/5010>

¹³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/5009>

¹³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/5008>

¹³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/5007>

¹³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/5006>

¹⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/5005>

¹⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/5004>

¹⁴⁰² <https://github.com/STELLAR-GROUP/hpx/pull/5003>

¹⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/5001>

¹⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4999>

- PR #4997¹⁴⁰⁵ - Fix unused variable and typedef warnings
- PR #4996¹⁴⁰⁶ - Adapt adjacent_find to C++20
- PR #4995¹⁴⁰⁷ - Test all schedulers in cross_pool_injection test except shared_priority_queue_scheduler
- PR #4993¹⁴⁰⁸ - Fix deprecation warnings
- PR #4991¹⁴⁰⁹ - Avoid unnecessarily including entire modules
- PR #4990¹⁴¹⁰ - Fixing some warnings from HPX complaining about use of obsolete types
- PR #4989¹⁴¹¹ - add a *destroy* trait for ParcelPort plugins
- PR #4986¹⁴¹² - Remove serialization to functional module dependency
- PR #4985¹⁴¹³ - Compatibility header generation
- PR #4980¹⁴¹⁴ - Add ranges overloads to for_loop (and variants)
- PR #4979¹⁴¹⁵ - Actually enable unity builds on Jenkins
- PR #4977¹⁴¹⁶ - Cleaning up debug::print functionalities
- PR #4976¹⁴¹⁷ - Remove indirection layer in at_index_impl
- PR #4975¹⁴¹⁸ - Remove indirection layer in at_index_impl
- PR #4973¹⁴¹⁹ - Avoid warnings/errors for older gcc complaining about multi-line comments
- PR #4970¹⁴²⁰ - Making set algorithms conform to C++20
- PR #4969¹⁴²¹ - Moving is_execution_policy and friends into namespace hpx
- PR #4968¹⁴²² - Enable deprecation warnings for 1.6.0 and move any functionality to hpx namespace
- PR #4967¹⁴²³ - Define deprecation macros conditionally
- PR #4966¹⁴²⁴ - Add clang-format and cmake-format version prints
- PR #4965¹⁴²⁵ - Making is_heap and is_heap_until conforming to C++20
- PR #4964¹⁴²⁶ - Adding parallel make_heap
- PR #4962¹⁴²⁷ - Fix external timer function pointer exports

¹⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4997>

¹⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4996>

¹⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4995>

¹⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4993>

¹⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4991>

¹⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4990>

¹⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4989>

¹⁴¹² <https://github.com/STELLAR-GROUP/hpx/pull/4986>

¹⁴¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4985>

¹⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4980>

¹⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4979>

¹⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4977>

¹⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4976>

¹⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4975>

¹⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4973>

¹⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4970>

¹⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4969>

¹⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/4968>

¹⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/4967>

¹⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4966>

¹⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4965>

¹⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4964>

¹⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4962>

- PR #4960¹⁴²⁸ - Fixing folder names for module tests and examples
- PR #4959¹⁴²⁹ - Adding communications set
- PR #4958¹⁴³⁰ - Deprecate tuple and timing functionality in `hpx::util`
- PR #4957¹⁴³¹ - Fixing unity build option for parcelports
- PR #4953¹⁴³² - Fixing MSVC problems after recent restructurings
- PR #4952¹⁴³³ - Make `parallel_executor` use `thread_pool_executor` spawning mechanism
- PR #4948¹⁴³⁴ - Clean up old artifacts better and more aggressively on Jenkins
- PR #4947¹⁴³⁵ - Add HIP support for AMD GPUs
- PR #4945¹⁴³⁶ - Enable `HPX_WITH_UNITY_BUILD` option on one of the Jenkins configurations
- PR #4943¹⁴³⁷ - Move public `hpx::parallel::execution` functionality to `hpx::execution`
- PR #4938¹⁴³⁸ - Post release cleanup
- PR #4858¹⁴³⁹ - Extending resilience APIs to support distributed invocations
- PR #4744¹⁴⁴⁰ - Fork-join executor
- PR #4665¹⁴⁴¹ - Implementing sender, receiver, and `operation_state` concepts in terms of P0443r13
- PR #4649¹⁴⁴² - Split libhpx into multiple libraries
- PR #4642¹⁴⁴³ - Implementing `operation_state` concept in terms of P0443r13
- PR #4640¹⁴⁴⁴ - Implementing receiver concept in terms of P0443r13
- PR #4622¹⁴⁴⁵ - Sanitizer fixes

2.10.7 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.

¹⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4960>

¹⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4959>

¹⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4958>

¹⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4957>

¹⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/4953>

¹⁴³³ <https://github.com/STELLAR-GROUP/hpx/pull/4952>

¹⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4948>

¹⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4947>

¹⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4945>

¹⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4943>

¹⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4938>

¹⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4858>

¹⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4744>

¹⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4665>

¹⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4649>

¹⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4642>

¹⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4640>

¹⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4622>

Closed issues

- Issue #4971¹⁴⁴⁶ - Parallel sort fails to compile with C++20
- Issue #4950¹⁴⁴⁷ - Build with *HPX_WITH_PARCELPORT_ACTION_COUNTERS ON* fails
- Issue #4940¹⁴⁴⁸ - Codespell report for “HPX” (on fossies.org)
- Issue #4937¹⁴⁴⁹ - Allow suspension of runtime for multiple localities

Closed pull requests

- PR #4982¹⁴⁵⁰ - Add page about citing HPX to documentation
- PR #4981¹⁴⁵¹ - Adding the missing include
- PR #4974¹⁴⁵² - Remove leftover format export hack
- PR #4972¹⁴⁵³ - Removing use of `get_temporary_buffer` and `return_temporary_buffer`
- PR #4963¹⁴⁵⁴ - Renaming files to avoid warnings from the vs build system
- PR #4951¹⁴⁵⁵ - Fixing build if `HPX_WITH_PARCELPORT_ACTION_COUNTERS=On`
- PR #4946¹⁴⁵⁶ - Allow suspension on multiple localities
- PR #4944¹⁴⁵⁷ - Fix typos reported by fossies codespell report
- PR #4941¹⁴⁵⁸ - Adding some explanation to README about how to cite HPX
- PR #4939¹⁴⁵⁹ - Small changes

2.10.8 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.

¹⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4971>

¹⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4950>

¹⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4940>

¹⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4937>

¹⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4982>

¹⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4981>

¹⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4974>

¹⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4972>

¹⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4963>

¹⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4951>

¹⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4946>

¹⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4944>

¹⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4941>

¹⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4939>

- We have adapted existing synchronization primitives to be fully conformant C++20: `hpx::barrier`, `hpx::latch`, `hpx::counting_semaphore`, and `hpx::binary_semaphore`.
- We have started using customization point objects (CPOs) to make the corresponding algorithms fully conformant to C++20 as well as to make algorithm extension easier for the user. `all_of/any_of/none_of`, `copy`, `count`, `destroy`, `equal`, `fill`, `find`, `for_each`, `generate`, `mismatch`, `move`, `reduce`, `transform_reduce` are using those CPOs (all in namespace `hpx`). We also have adapted their corresponding `hpx::ranges` versions to be conforming to C++20 in this release.
- We have adapted support for `co_await` to C++20, in addition to `hpx::future` it now also supports `hpx::shared_future`. We have also added allocator support for futures returned by `co_return`. It is no longer in the `experimental` namespace.
- We added serialization support for `std::variant` and `std::tuple`.
- `result_of` and `is_callable` are now deprecated and replaced by `invoke_result` and `is_invocable` to conform to C++20.
- We continued with the modularization, making it easier for us to add the new experimental `HPX_WITH_DISTRIBUTED_RUNTIME` CMake option (see below). An significant amount of headers have been deprecated. We adapted the namespaces and headers we could to be closer to the standard ones (*Public API*). Depending code should still compile, however warnings are now generated instructing to change the include statements accordingly.
- It is now possible to have a basic CUDA support including a helper function to get a future from a CUDA stream and target handling. They are available under the `hpx::cuda::experimental` namespace and they can be enabled with the `-DHPX_WITH_ASYNC_CUDA=ON` CMake option.
- We added a new `hpx::mpi::experimental` namespace for getting futures from an asynchronous MPI call and a new minimal MPI executor `hpx::mpi::experimental::executor`. These can be enabled with the `-DHPX_WITH_ASYNC_MPI=On` CMake option.
- A polymorphic executor has been implemented to reduce compile times as a function accepting executors can potentially be instantiated only once instead of multiple times with different executors. It accepts the function signature as a template argument. It needs to be constructed from any other executor. Please note, that the function signatures that can be scheduled using `then_execute`, `bulk_sync_execute`, `bulk_async_execute` and `bulk_then_execute` are slightly different (See the comment in PR #4514¹⁴⁶⁰ for more details).
- The underlying executor of `block_executor` has been updated to a newer one.
- We have added a parameter to `auto_chunk_size` to control the amount of iterations to measure.
- All executor parameter hooks can now be exposed through the executor itself. This will allow to deprecate the `.with()` functionality on execution policies in the future. This is also a first step towards simplifying our executor APIs in preparation for the upcoming C++23 executors (senders/receivers).
- We have moved all of the existing APIs related to resiliency into the namespace `hpx::resiliency::experimental`. Please note this is a breaking change without backwards-compatibility option. We have converted all of those APIs to be based on customization point objects. Two new executors have been added to enable easy integration of the existing resiliency features with other facilities (like the parallel algorithms): `replay_executor` and `replicate_executor`.
- We have added performance counters type information (`aggregating`, `monotonically increasing`, `average count`, `average timer`, etc.).
- HPX threads are now re-scheduled on the same worker thread they were suspended on to avoid cache misses from moving from one thread to the other. This behavior doesn't prevent the thread from being stolen, however.
- We have added a new configuration option `hpx.exception_verbosity` to allow to control the level of verbosity of the exceptions (3 levels available).

¹⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4514>

- `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 #4918¹⁴⁶¹ - Rename distributed_executors module
- Issue #4900¹⁴⁶² - Adding JOSS status badge to README
- Issue #4897¹⁴⁶³ - Compiler warning, deprecated header used by HPX itself
- Issue #4886¹⁴⁶⁴ - A future bound to an action executing on a different locality doesn't capture exception state
- Issue #4880¹⁴⁶⁵ - Undefined reference to main build error when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- Issue #4877¹⁴⁶⁶ - hpx_main might not able to start hpx runtime properly
- Issue #4850¹⁴⁶⁷ - Issues creating templated component
- Issue #4829¹⁴⁶⁸ - Spack package & HPX_WITH_GENERIC_CONTEXT_COROUTINES
- Issue #4820¹⁴⁶⁹ - PAPI counters don't work
- Issue #4818¹⁴⁷⁰ - HPX can't be used with IO pool turned off
- Issue #4816¹⁴⁷¹ - Build of HPX fails when find_package(Boost) is called before FetchContent_MakeAvailable(hpx)
- Issue #4813¹⁴⁷² - HPX MPI Future failed
- Issue #4811¹⁴⁷³ - Remove HPX::hpx_no_wrap_main target before 1.5.0 release
- Issue #4810¹⁴⁷⁴ - In hpx::for_each::invoke_projected the hpx::util::decay is misguided
- Issue #4787¹⁴⁷⁵ - transform_inclusive_scan gives incorrect results for non-commutative operator
- Issue #4786¹⁴⁷⁶ - transform_inclusive_scan tries to implicitly convert between types, instead of using the provided conv function
- Issue #4779¹⁴⁷⁷ - HPX build error with GCC 10.1
- Issue #4766¹⁴⁷⁸ - Move HPX.Compute functionality to experimental namespace
- Issue #4763¹⁴⁷⁹ - License file name
- Issue #4758¹⁴⁸⁰ - CMake profiling results
- Issue #4755¹⁴⁸¹ - Building HPX with support for PAPI fails

¹⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/4918>

¹⁴⁶² <https://github.com/STELLAR-GROUP/hpx/issues/4900>

¹⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/4897>

¹⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4886>

¹⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4880>

¹⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4877>

¹⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4850>

¹⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4829>

¹⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4820>

¹⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4818>

¹⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/4816>

¹⁴⁷² <https://github.com/STELLAR-GROUP/hpx/issues/4813>

¹⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/4811>

¹⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4810>

¹⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4787>

¹⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4786>

¹⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4779>

¹⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4766>

¹⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4763>

¹⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4758>

¹⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/4755>

- Issue #4754¹⁴⁸² - CMake cache creation breaks when using HPX with mimalloc
- Issue #4752¹⁴⁸³ - HPX MPI Future build failed
- Issue #4746¹⁴⁸⁴ - Memory leak when using dataflow icw components
- Issue #4731¹⁴⁸⁵ - Bug in stencil example, calculation of locality IDs
- Issue #4723¹⁴⁸⁶ - Build fail with NETWORKING OFF
- Issue #4720¹⁴⁸⁷ - Add compatibility headers for modules that had their module headers implicitly generated in 1.4.1
- Issue #4719¹⁴⁸⁸ - Undeprecate some module headers
- Issue #4712¹⁴⁸⁹ - Rename HPX_MPI_WITH_FUTURES option
- Issue #4709¹⁴⁹⁰ - Make deprecation warnings overridable in dependent projects
- Issue #4691¹⁴⁹¹ - Suggestion to fix and enhance the thread_mapper API
- Issue #4686¹⁴⁹² - Fix tutorials examples
- Issue #4685¹⁴⁹³ - HPX distributed map fails to compile
- Issue #4680¹⁴⁹⁴ - Build error with HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- Issue #4679¹⁴⁹⁵ - Build error for hpx w/ Apex on Summit
- Issue #4675¹⁴⁹⁶ - build error with HPX_WITH_NETWORKING=OFF
- Issue #4674¹⁴⁹⁷ - Error running Quickstart tests on OS X
- Issue #4662¹⁴⁹⁸ - MPI initialization broken when networking off
- Issue #4652¹⁴⁹⁹ - How to fix distributed action annotation
- Issue #4650¹⁵⁰⁰ - thread descriptions are broken... again
- Issue #4648¹⁵⁰¹ - Thread stacksize not properly set
- Issue #4647¹⁵⁰² - Rename generated collective headers in modules
- Issue #4639¹⁵⁰³ - Update deprecation warnings in compatibility headers to point to collective headers
- Issue #4628¹⁵⁰⁴ - mpi parcelport totally broken

¹⁴⁸² <https://github.com/STELLAR-GROUP/hpx/issues/4754>

¹⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/4752>

¹⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4746>

¹⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4731>

¹⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4723>

¹⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4720>

¹⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4719>

¹⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4712>

¹⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4709>

¹⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/4691>

¹⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/4686>

¹⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/4685>

¹⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4680>

¹⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4679>

¹⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4675>

¹⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4674>

¹⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4662>

¹⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4652>

¹⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4650>

¹⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/4648>

¹⁵⁰² <https://github.com/STELLAR-GROUP/hpx/issues/4647>

¹⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/4639>

¹⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4628>

- Issue #4619¹⁵⁰⁵ - Fully document hpx_wrap behaviour and targets
- Issue #4612¹⁵⁰⁶ - Compilation issue with HPX 1.4.1 and 1.4.0
- Issue #4594¹⁵⁰⁷ - Rename modules
- Issue #4578¹⁵⁰⁸ - Default value for HPX_WITH_THREAD_BACKTRACE_DEPTH
- Issue #4572¹⁵⁰⁹ - Thread manager should be given a runtime_configuration
- Issue #4571¹⁵¹⁰ - Add high-level documentation to new modules
- Issue #4569¹⁵¹¹ - Annoying warning when compiling - pls suppress or fix it.
- Issue #4555¹⁵¹² - HPX_HAVE_THREAD_BACKTRACE_ON_SUSPENSION compilation error
- Issue #4543¹⁵¹³ - Segfaults in Release builds using *sleep_for*
- Issue #4539¹⁵¹⁴ - Compilation Error when HPX_MPI_WITH_FUTURES=ON
- Issue #4537¹⁵¹⁵ - Linking issue with libhpx_initd.a
- Issue #4535¹⁵¹⁶ - API for checking if pool with a given name exists
- Issue #4523¹⁵¹⁷ - Build of PR #4311 (git tag 9955e8e) fails
- Issue #4519¹⁵¹⁸ - Documentation problem
- Issue #4513¹⁵¹⁹ - HPXConfig.cmake contains ill-formed paths when library paths use backslashes
- Issue #4507¹⁵²⁰ - User-polling introduced by MPI futures module should be more generally usable
- Issue #4506¹⁵²¹ - Make sure force_linking.hpp is not included in main module header
- Issue #4501¹⁵²² - Fix compilation of PAPI tests
- Issue #4497¹⁵²³ - Add modules CI checks
- Issue #4489¹⁵²⁴ - Polymorphic executor
- Issue #4476¹⁵²⁵ - Use CMake targets defined by FindBoost
- Issue #4473¹⁵²⁶ - Add vcpkg installation instructions
- Issue #4470¹⁵²⁷ - Adapt hpx::future to C++20 co_await

¹⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4619>

¹⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4612>

¹⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4594>

¹⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4578>

¹⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4572>

¹⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4571>

1511 <https://github.com/STELLAR-GROUP/hpx/issues/4569>1512 <https://github.com/STELLAR-GROUP/hpx/issues/4555>1513 <https://github.com/STELLAR-GROUP/hpx/issues/4543>1514 <https://github.com/STELLAR-GROUP/hpx/issues/4539>1515 <https://github.com/STELLAR-GROUP/hpx/issues/4537>1516 <https://github.com/STELLAR-GROUP/hpx/issues/4535>1517 <https://github.com/STELLAR-GROUP/hpx/issues/4523>1518 <https://github.com/STELLAR-GROUP/hpx/issues/4519>1519 <https://github.com/STELLAR-GROUP/hpx/issues/4513>1520 <https://github.com/STELLAR-GROUP/hpx/issues/4507>1521 <https://github.com/STELLAR-GROUP/hpx/issues/4506>1522 <https://github.com/STELLAR-GROUP/hpx/issues/4501>1523 <https://github.com/STELLAR-GROUP/hpx/issues/4497>1524 <https://github.com/STELLAR-GROUP/hpx/issues/4489>1525 <https://github.com/STELLAR-GROUP/hpx/issues/4476>1526 <https://github.com/STELLAR-GROUP/hpx/issues/4473>1527 <https://github.com/STELLAR-GROUP/hpx/issues/4470>

- Issue #4468¹⁵²⁸ - Compile error on Raspberry Pi 4
- Issue #4466¹⁵²⁹ - Compile error on Windows, current stable:
- Issue #4453¹⁵³⁰ - Installing HPX on fedora with dnf is not adding cmake files
- Issue #4448¹⁵³¹ - New std::variant serialization broken
- Issue #4438¹⁵³² - Add performance counter flag is monotonically increasing
- Issue #4436¹⁵³³ - Build problem: same code build and works with 1.4.0 but it doesn't with 1.4.1
- Issue #4429¹⁵³⁴ - Function descriptions not supported in distributed
- Issue #4423¹⁵³⁵ - --hpx:ini=hpx.lock_detection=0 has no effect
- Issue #4422¹⁵³⁶ - Add performance counter metadata
- Issue #4419¹⁵³⁷ - Weird behavior for --hpx:print-counter-interval with large numbers
- Issue #4401¹⁵³⁸ - Create module repository
- Issue #4400¹⁵³⁹ - Command line options conflict related to performance counters
- Issue #4349¹⁵⁴⁰ - --hpx:use-process-mask option throw an exception on OS X
- Issue #4345¹⁵⁴¹ - Move gh-pages branch out of hpx repo
- Issue #4323¹⁵⁴² - Const-correctness error in assignment operator of compute::vector
- Issue #4318¹⁵⁴³ - ASIO breaks with C++2a concepts
- Issue #4317¹⁵⁴⁴ - Application runs even if --hpx:help is specified
- Issue #4063¹⁵⁴⁵ - Document hpxcxx compiler wrapper
- Issue #3983¹⁵⁴⁶ - Implement the C++20 Synchronization Library
- Issue #3696¹⁵⁴⁷ - C++11 *constexpr* support is now required
- Issue #3623¹⁵⁴⁸ - Modular HPX branch and an alternative project layout
- Issue #2836¹⁵⁴⁹ - The worst-case time complexity of parallel::sort seems to be O(N^2).

1528 <https://github.com/STELLAR-GROUP/hpx/issues/4468>

1529 <https://github.com/STELLAR-GROUP/hpx/issues/4466>

1530 <https://github.com/STELLAR-GROUP/hpx/issues/4453>

1531 <https://github.com/STELLAR-GROUP/hpx/issues/4448>

1532 <https://github.com/STELLAR-GROUP/hpx/issues/4438>

1533 <https://github.com/STELLAR-GROUP/hpx/issues/4436>

1534 <https://github.com/STELLAR-GROUP/hpx/issues/4429>

1535 <https://github.com/STELLAR-GROUP/hpx/issues/4423>

1536 <https://github.com/STELLAR-GROUP/hpx/issues/4422>

1537 <https://github.com/STELLAR-GROUP/hpx/issues/4419>

1538 <https://github.com/STELLAR-GROUP/hpx/issues/4401>

1539 <https://github.com/STELLAR-GROUP/hpx/issues/4400>

1540 <https://github.com/STELLAR-GROUP/hpx/issues/4349>

1541 <https://github.com/STELLAR-GROUP/hpx/issues/4345>

1542 <https://github.com/STELLAR-GROUP/hpx/issues/4323>

1543 <https://github.com/STELLAR-GROUP/hpx/issues/4318>

1544 <https://github.com/STELLAR-GROUP/hpx/issues/4317>

1545 <https://github.com/STELLAR-GROUP/hpx/issues/4063>

1546 <https://github.com/STELLAR-GROUP/hpx/issues/3983>

1547 <https://github.com/STELLAR-GROUP/hpx/issues/3696>

1548 <https://github.com/STELLAR-GROUP/hpx/issues/3623>

1549 <https://github.com/STELLAR-GROUP/hpx/issues/2836>

Closed pull requests

- PR #4936¹⁵⁵⁰ - Minor documentation fixes part 2
- PR #4935¹⁵⁵¹ - Add copyright and license to joss paper file
- PR #4934¹⁵⁵² - Adding Semicolon in Documentation
- PR #4932¹⁵⁵³ - Fixing compiler warnings
- PR #4931¹⁵⁵⁴ - Small documentation formatting fixes
- PR #4930¹⁵⁵⁵ - Documentation Distributed HPX applications localvv with local_vv
- PR #4929¹⁵⁵⁶ - Add final version of the JOSS paper
- PR #4928¹⁵⁵⁷ - Add HPX_NODISCARD to enable_user_polling structs
- PR #4926¹⁵⁵⁸ - Rename distributed_executors module to executors_distributed
- PR #4925¹⁵⁵⁹ - Making transform_reduce conforming to C++20
- PR #4923¹⁵⁶⁰ - Don't acquire lock if not needed
- PR #4921¹⁵⁶¹ - Update the release notes for the release candidate 3
- PR #4920¹⁵⁶² - Disable libcds release
- PR #4919¹⁵⁶³ - Make cuda event pool dynamic instead of fixed size
- PR #4917¹⁵⁶⁴ - Move chrono functionality to hpx::chrono namespace
- PR #4916¹⁵⁶⁵ - HPX_HAVE_DEPRECATED_WARNINGS needs to be set even when disabled
- PR #4915¹⁵⁶⁶ - Moving more action related files to actions modules
- PR #4914¹⁵⁶⁷ - Add alias targets with namespaces used for exporting
- PR #4912¹⁵⁶⁸ - Aggregate initialize CPOs
- PR #4910¹⁵⁶⁹ - Explicitly specify hwloc root on Jenkins CSCS builds
- PR #4908¹⁵⁷⁰ - Fix algorithms documentation
- PR #4907¹⁵⁷¹ - Remove HPX::hpx_no_wrap_main target

¹⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4936>

¹⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4935>

¹⁵⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4934>

¹⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4932>

¹⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4931>

¹⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4930>

¹⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4929>

¹⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4928>

¹⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4926>

¹⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4925>

¹⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4923>

¹⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4921>

¹⁵⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4920>

¹⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4919>

¹⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4917>

¹⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4916>

¹⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4915>

¹⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4914>

¹⁵⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4912>

¹⁵⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4910>

¹⁵⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4908>

¹⁵⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4907>

- PR #4906¹⁵⁷² - Fixing unused variable warning
- PR #4905¹⁵⁷³ - Adding specializations for simple for_loops
- PR #4904¹⁵⁷⁴ - Update boost to 1.74.0 for the newest jenkins configs
- PR #4903¹⁵⁷⁵ - Hide GITHUB_TOKEN environment variables from environment variable output
- PR #4902¹⁵⁷⁶ - Cancel previous pull requests builds before starting a new one with Jenkins
- PR #4901¹⁵⁷⁷ - Update public API list with updated algorithms
- PR #4899¹⁵⁷⁸ - Suggested changes for HPX V1.5 release notes
- PR #4898¹⁵⁷⁹ - Minor tweak to hpx::equal implementation
- PR #4896¹⁵⁸⁰ - Making generate() and generate_n conforming to C++20
- PR #4895¹⁵⁸¹ - Update apex tag
- PR #4894¹⁵⁸² - Fix exception handling for tasks
- PR #4893¹⁵⁸³ - Remove last use of std::result_of, removed in C++20
- PR #4892¹⁵⁸⁴ - Adding replay_executor and replicate_executor
- PR #4889¹⁵⁸⁵ - Restore old behaviour of not requiring linking to hpx_wrap when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4887¹⁵⁸⁶ - Making sure remotely thrown (non-hpx) exceptions are properly marshaled back to invocation site
- PR #4885¹⁵⁸⁷ - Adapting hpx::find and friends to C++20
- PR #4884¹⁵⁸⁸ - Adapting mismatch to C++20
- PR #4883¹⁵⁸⁹ - Adapting hpx::equal to be conforming to C++20
- PR #4882¹⁵⁹⁰ - Fixing exception handling for hpx::copy and adding missing tests
- PR #4881¹⁵⁹¹ - Adds different runtime exception when registering thread with the HPX runtime
- PR #4876¹⁵⁹² - Adding example demonstrating how to disable thread stealing during the execution of parallel algorithms
- PR #4874¹⁵⁹³ - Adding non-policy tests to all_of, any_of, and none_of

¹⁵⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4906>

¹⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4905>

¹⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4904>

¹⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4903>

¹⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4902>

¹⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4901>

¹⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4899>

1579 <https://github.com/STELLAR-GROUP/hpx/pull/4898>1580 <https://github.com/STELLAR-GROUP/hpx/pull/4896>1581 <https://github.com/STELLAR-GROUP/hpx/pull/4895>1582 <https://github.com/STELLAR-GROUP/hpx/pull/4894>1583 <https://github.com/STELLAR-GROUP/hpx/pull/4893>1584 <https://github.com/STELLAR-GROUP/hpx/pull/4892>1585 <https://github.com/STELLAR-GROUP/hpx/pull/4889>1586 <https://github.com/STELLAR-GROUP/hpx/pull/4887>1587 <https://github.com/STELLAR-GROUP/hpx/pull/4885>1588 <https://github.com/STELLAR-GROUP/hpx/pull/4884>1589 <https://github.com/STELLAR-GROUP/hpx/pull/4883>1590 <https://github.com/STELLAR-GROUP/hpx/pull/4882>1591 <https://github.com/STELLAR-GROUP/hpx/pull/4881>1592 <https://github.com/STELLAR-GROUP/hpx/pull/4876>1593 <https://github.com/STELLAR-GROUP/hpx/pull/4874>

- PR #4873¹⁵⁹⁴ - Set CUDA compute capability on rostam Jenkins builds
- PR #4872¹⁵⁹⁵ - Force partitioned vector scan tests to run serially
- PR #4870¹⁵⁹⁶ - Making move conforming with C++20
- PR #4869¹⁵⁹⁷ - Making destroy and destroy_n conforming to C++20
- PR #4868¹⁵⁹⁸ - Fix miscellaneous header problems
- PR #4867¹⁵⁹⁹ - Add CPOs for for_each
- PR #4865¹⁶⁰⁰ - Adapting count and count_if to be conforming to C++20
- PR #4864¹⁶⁰¹ - Release notes 1.5.0
- PR #4863¹⁶⁰² - adding libcds-hpx tag to prepare for hpx 1.5 release
- PR #4862¹⁶⁰³ - Adding version specific deprecation options
- PR #4861¹⁶⁰⁴ - Limiting executor improvements
- PR #4860¹⁶⁰⁵ - Making fill and fill_n compatible with C++20
- PR #4859¹⁶⁰⁶ - Adapting all_of, any_of, and none_of to C++20
- PR #4857¹⁶⁰⁷ - Improve libCDS integration
- PR #4856¹⁶⁰⁸ - Correct typos in the documentation of the hpx performance counters
- PR #4854¹⁶⁰⁹ - Removing obsolete code
- PR #4853¹⁶¹⁰ - Adding test that derives component from two other components
- PR #4852¹⁶¹¹ - Fix mpi_ring test in distributed mode by ensuring all ranks run hpx_main
- PR #4851¹⁶¹² - Converting resiliency APIs to tag_invoke based CPOs
- PR #4849¹⁶¹³ - Enable use of future_overhead test when DISTRIBUTED_RUNTIME is OFF
- PR #4847¹⁶¹⁴ - Fixing ‘error prone’ constructs as reported by Codacy
- PR #4846¹⁶¹⁵ - Disable Boost.Aasio concepts support
- PR #4845¹⁶¹⁶ - Fix PAPI counters

¹⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4873>

¹⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4872>

¹⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4870>

¹⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4869>

¹⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4868>

¹⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4867>

¹⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4865>

¹⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4864>

¹⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4863>

¹⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4862>

¹⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4861>

¹⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4860>

¹⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4859>

¹⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4857>

¹⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4856>

¹⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4854>

¹⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4853>

¹⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4852>

¹⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/4851>

¹⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4849>

¹⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4847>

¹⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4846>

¹⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4845>

- PR #4843¹⁶¹⁷ - Remove dependency on various Boost headers
- PR #4841¹⁶¹⁸ - Rearrange public API headers
- PR #4840¹⁶¹⁹ - Fixing TSS problems during thread termination
- PR #4839¹⁶²⁰ - Fix async_cuda build problems when distributed runtime is disabled
- PR #4837¹⁶²¹ - Restore compatibility for old (now deprecated) copy algorithms
- PR #4836¹⁶²² - Adding CPOs for hpx::reduce
- PR #4835¹⁶²³ - Remove *using util::result_of* from namespace hpx
- PR #4834¹⁶²⁴ - Fixing the calculation of the number of idle cores and the corresponding idle masks
- PR #4833¹⁶²⁵ - Allow thread function destructors to yield
- PR #4832¹⁶²⁶ - Fixing assertion in split_gids and memory leaks in 1d_stencil_7
- PR #4831¹⁶²⁷ - Making sure MPI_CXX_COMPILE_FLAGS is interpreted as a sequence of options
- PR #4830¹⁶²⁸ - Update documentation on using HPX::wrap_main
- PR #4827¹⁶²⁹ - Update clang-newest configuration to use clang 10
- PR #4826¹⁶³⁰ - Add Jenkins configuration for rostam
- PR #4825¹⁶³¹ - Move all CUDA functionality to hpx::cuda::experimental namespace
- PR #4824¹⁶³² - Add support for building master/release branches to Jenkins configuration
- PR #4821¹⁶³³ - Implement customization point for hpx::copy and hpx::ranges::copy
- PR #4819¹⁶³⁴ - Allow finding Boost components before finding HPX
- PR #4817¹⁶³⁵ - Adding range version of stable sort
- PR #4815¹⁶³⁶ - Fix a wrong #ifdef for IO/TIMER pools causing build errors
- PR #4814¹⁶³⁷ - Replace hpx::function_nonser with std::function in error module
- PR #4809¹⁶³⁸ - Foreach adapt
- PR #4808¹⁶³⁹ - Make internal algorithms functions const

¹⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4843>

¹⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4841>

¹⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4840>

¹⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4839>

¹⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4837>

¹⁶²² <https://github.com/STELLAR-GROUP/hpx/pull/4836>

¹⁶²³ <https://github.com/STELLAR-GROUP/hpx/pull/4835>

¹⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4834>

¹⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4833>

¹⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4832>

¹⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4831>

¹⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4830>

¹⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4827>

¹⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4826>

¹⁶³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4825>

¹⁶³² <https://github.com/STELLAR-GROUP/hpx/pull/4824>

¹⁶³³ <https://github.com/STELLAR-GROUP/hpx/pull/4821>

¹⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4819>

¹⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4817>

¹⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4815>

¹⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4814>

¹⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4809>

¹⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4808>

- PR #4807¹⁶⁴⁰ - Add Jenkins configuration for running on Piz Daint
- PR #4806¹⁶⁴¹ - Update documentation links to new domain name
- PR #4805¹⁶⁴² - Applying changes that resolve time complexity issues in sort
- PR #4803¹⁶⁴³ - Adding implementation of stable_sort
- PR #4802¹⁶⁴⁴ - Fix datapar header paths
- PR #4801¹⁶⁴⁵ - Replace boost::shared_array<T> with std::shared_ptr<T[]> if supported
- PR #4799¹⁶⁴⁶ - Fixing #include paths in compatibility headers
- PR #4798¹⁶⁴⁷ - Include the main module header (fixes partially #4488)
- PR #4797¹⁶⁴⁸ - Change cmake targets
- PR #4794¹⁶⁴⁹ - Removing 128bit integer emulation
- PR #4793¹⁶⁵⁰ - Make sure global variable is handled properly
- PR #4792¹⁶⁵¹ - Replace enable_if with **HPX_CONCEPT_REQUIRE**s and add is_sentinel_for constraint
- PR #4790¹⁶⁵² - Move deprecation warnings from base template to template specializations for result_of etc. structs
- PR #4789¹⁶⁵³ - Fix hangs during assertion handling and distributed runtime construction
- PR #4788¹⁶⁵⁴ - Fixing inclusive transform scan algorithm to properly handle initial value
- PR #4785¹⁶⁵⁵ - Fixing barrier test
- PR #4784¹⁶⁵⁶ - Fixing deleter argument bindings in serialize_buffer
- PR #4783¹⁶⁵⁷ - Add coveralls badge
- PR #4782¹⁶⁵⁸ - Make header tests parallel again
- PR #4780¹⁶⁵⁹ - Remove outdated comment about hpx::stop in documentation
- PR #4776¹⁶⁶⁰ - debug print improvements
- PR #4775¹⁶⁶¹ - Checkpoint cleanup
- PR #4771¹⁶⁶² - Fix compilation with HPX_WITH_NETWORKING=OFF

¹⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4807>

¹⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4806>

¹⁶⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4805>

¹⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4803>

¹⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4802>

¹⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4801>

¹⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4799>

¹⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4798>

¹⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4797>

¹⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4794>

¹⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4793>

¹⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4792>

¹⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4790>

¹⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4789>

¹⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4788>

¹⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4785>

¹⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4784>

¹⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4783>

¹⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4782>

¹⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4780>

¹⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4776>

¹⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4775>

¹⁶⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4771>

- PR #4767¹⁶⁶³ - Remove all force linking leftovers
- PR #4765¹⁶⁶⁴ - Fix 1d stencil index calculation
- PR #4764¹⁶⁶⁵ - Force some tests to run serially
- PR #4762¹⁶⁶⁶ - Update pointees in compatibility headers
- PR #4761¹⁶⁶⁷ - Fix running and building of execution module tests on CircleCI
- PR #4760¹⁶⁶⁸ - Storing hpx_options in global property to speed up summary report
- PR #4759¹⁶⁶⁹ - Reduce memory requirements for our main shared state
- PR #4757¹⁶⁷⁰ - Fix mimalloc linking on Windows
- PR #4756¹⁶⁷¹ - Fix compilation issues
- PR #4753¹⁶⁷² - Re-adding API functions that were lost during merges
- PR #4751¹⁶⁷³ - Revert “Create coverage reports and upload them to codecov.io”
- PR #4750¹⁶⁷⁴ - Fixing possible race condition during termination detection
- PR #4749¹⁶⁷⁵ - Deprecate result_of and friends
- PR #4748¹⁶⁷⁶ - Create coverage reports and upload them to codecov.io
- PR #4747¹⁶⁷⁷ - Changing #include for MPI parcelport
- PR #4745¹⁶⁷⁸ - Add *is_sentinel_for* trait implementation and test
- PR #4743¹⁶⁷⁹ - Fix init_globally example after runtime mode changes
- PR #4742¹⁶⁸⁰ - Update SUPPORT.md
- PR #4741¹⁶⁸¹ - Fixing a warning generated for unity builds with msvc
- PR #4740¹⁶⁸² - Rename local_lcos and basic_execution modules
- PR #4739¹⁶⁸³ - Undeprecate a couple of hpx/modulename.hpp headers
- PR #4738¹⁶⁸⁴ - Conditionally test schedulers in thread_stacksize_current test
- PR #4734¹⁶⁸⁵ - Fixing a bunch of codacy warnings

¹⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4767>

¹⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4765>

¹⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4764>

¹⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4762>

¹⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4761>

¹⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4760>

¹⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4759>

¹⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4757>

¹⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4756>

¹⁶⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4753>

¹⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4751>

¹⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4750>

¹⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4749>

¹⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4748>

¹⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4747>

¹⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4745>

¹⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4743>

¹⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4742>

¹⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4741>

¹⁶⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4740>

¹⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4739>

¹⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4738>

¹⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4734>

- PR #4733¹⁶⁸⁶ - Add experimental unity build option to CMake configuration
- PR #4730¹⁶⁸⁷ - Fixing compilation problems with unordered map
- PR #4729¹⁶⁸⁸ - Fix APEX build
- PR #4727¹⁶⁸⁹ - Fix missing runtime includes for distributed runtime
- PR #4726¹⁶⁹⁰ - Add more API headers
- PR #4725¹⁶⁹¹ - Add more compatibility headers for deprecated module headers
- PR #4724¹⁶⁹² - Fix 4723
- PR #4721¹⁶⁹³ - Attempt to fixing migration tests
- PR #4717¹⁶⁹⁴ - Make the compatibility headers macro conditional
- PR #4716¹⁶⁹⁵ - Add hpx/runtime.hpp and hpx/distributed/runtime.hpp API headers
- PR #4714¹⁶⁹⁶ - Add hpx/future.hpp header
- PR #4713¹⁶⁹⁷ - Remove hpx/runtime/threads_fwd.hpp and hpx/util_fwd.hpp
- PR #4711¹⁶⁹⁸ - Make module deprecation warnings overridable
- PR #4710¹⁶⁹⁹ - Add compatibility headers and other fixes after module header renaming
- PR #4708¹⁷⁰⁰ - Add termination handler for parallel algorithms
- PR #4707¹⁷⁰¹ - Use hpx::function_nonser instead of std::function internally
- PR #4706¹⁷⁰² - Move header file to module
- PR #4705¹⁷⁰³ - Fix incorrect behaviour of cmake-format check
- PR #4704¹⁷⁰⁴ - Fix resource tests
- PR #4701¹⁷⁰⁵ - Fix missing includes for future::then specializations
- PR #4700¹⁷⁰⁶ - Removing obsolete memory component
- PR #4699¹⁷⁰⁷ - Add short descriptions to modules missing documentation
- PR #4696¹⁷⁰⁸ - Rename generated modules headers

¹⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4733>

¹⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4730>

¹⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4729>

¹⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4727>

¹⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4726>

¹⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4725>

¹⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4724>

¹⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4721>

¹⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4717>

¹⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4716>

¹⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4714>

¹⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4713>

¹⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4711>

¹⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4710>

¹⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4708>

¹⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4707>

¹⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4706>

¹⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4705>

1704 <https://github.com/STELLAR-GROUP/hpx/pull/4704>1705 <https://github.com/STELLAR-GROUP/hpx/pull/4701>1706 <https://github.com/STELLAR-GROUP/hpx/pull/4700>1707 <https://github.com/STELLAR-GROUP/hpx/pull/4699>1708 <https://github.com/STELLAR-GROUP/hpx/pull/4696>

- PR #4693¹⁷⁰⁹ - Overhauling thread_mapper for public consumption
- PR #4688¹⁷¹⁰ - Fix thread stack size handling
- PR #4687¹⁷¹¹ - Adding all_gather and fixing all_to_all
- PR #4684¹⁷¹² - Miscellaneous compilation fixes
- PR #4683¹⁷¹³ - Fix HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4682¹⁷¹⁴ - Fix compilation of pack_traversal_rebind_container.hpp
- PR #4681¹⁷¹⁵ - Add missing hpx/execution.hpp includes for future::then
- PR #4678¹⁷¹⁶ - Typeless communicator
- PR #4677¹⁷¹⁷ - Forcing registry option to be accepted without checks.
- PR #4676¹⁷¹⁸ - Adding scatter_to/scatter_from collective operations
- PR #4673¹⁷¹⁹ - Fix PAPI counters compilation
- PR #4671¹⁷²⁰ - Deprecate hpx::promise alias to hpx::lcos::promise
- PR #4670¹⁷²¹ - Explicitly instantiate get_exception
- PR #4667¹⁷²² - Add stopValue in *Sentinel* struct instead of *Iterator*
- PR #4666¹⁷²³ - Add release build on Windows to GitHub actions
- PR #4664¹⁷²⁴ - Creating itt_notify module.
- PR #4663¹⁷²⁵ - Mpi fixes
- PR #4659¹⁷²⁶ - Making sure declarations match definitions in register_locks implementation
- PR #4655¹⁷²⁷ - Fixing task annotations for actions
- PR #4653¹⁷²⁸ - Making sure APEX is linked into every application, if needed
- PR #4651¹⁷²⁹ - Update get_function_annotation.hpp
- PR #4646¹⁷³⁰ - Runtime type
- PR #4645¹⁷³¹ - Add a few more API headers

¹⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4693>

¹⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4688>

¹⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4687>

¹⁷¹² <https://github.com/STELLAR-GROUP/hpx/pull/4684>

¹⁷¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4683>

¹⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4682>

¹⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4681>

¹⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4678>

¹⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4677>

¹⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4676>

¹⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4673>

¹⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4671>

¹⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4670>

¹⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/4667>

¹⁷²³ <https://github.com/STELLAR-GROUP/hpx/pull/4666>

¹⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4664>

¹⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4663>

¹⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4659>

¹⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4655>

¹⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4653>

¹⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4651>

¹⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4646>

¹⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4645>

- PR #4644¹⁷³² - Fixing support for mpirun (and similar)
- PR #4643¹⁷³³ - Fixing the fix for get_idle_core_count() API
- PR #4638¹⁷³⁴ - Remove HPX_API_EXPORT missed in previous cleanup
- PR #4636¹⁷³⁵ - Adding C++20 barrier
- PR #4635¹⁷³⁶ - Adding C++20 latch API
- PR #4634¹⁷³⁷ - Adding C++20 counting semaphore API
- PR #4633¹⁷³⁸ - Unify execution parameters customization points
- PR #4632¹⁷³⁹ - Adding missing bulk_sync_execute wrapper to example executor
- PR #4631¹⁷⁴⁰ - Updates to documentation; grammar edits.
- PR #4630¹⁷⁴¹ - Updates to documentation; moved hyperlink
- PR #4624¹⁷⁴² - Export set_self_ptr in thread_data.hpp instead of with forward declarations where used
- PR #4623¹⁷⁴³ - Clean up export macros
- PR #4621¹⁷⁴⁴ - Trigger an error for older boost versions on power architectures
- PR #4617¹⁷⁴⁵ - Ignore user-set compatibility header options if the module does not have compatibility headers
- PR #4616¹⁷⁴⁶ - Fix cmake-format warning
- PR #4615¹⁷⁴⁷ - Add handler for serializing custom exceptions
- PR #4614¹⁷⁴⁸ - Fix error message when HPX_IGNORE_CMAKE_BUILD_TYPE_COMPATIBILITY=OFF
- PR #4613¹⁷⁴⁹ - Make partitioner constructor private
- PR #4611¹⁷⁵⁰ - Making auto_chunk_size execute the given function using the given executor
- PR #4610¹⁷⁵¹ - Making sure the thread-local lock registration data is moving to the core the suspended HPX thread is resumed on
- PR #4609¹⁷⁵² - Adding an API function that exposes the number of idle cores
- PR #4608¹⁷⁵³ - Fixing moodycamel namespace
- PR #4607¹⁷⁵⁴ - Moving winsocket initialization to core library

¹⁷³² <https://github.com/STELLAR-GROUP/hpx/pull/4644>

¹⁷³³ <https://github.com/STELLAR-GROUP/hpx/pull/4643>

¹⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4638>

¹⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4636>

¹⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4635>

¹⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4634>

¹⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4633>

¹⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4632>

¹⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4631>

¹⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4630>

¹⁷⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4624>

¹⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4623>

¹⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4621>

¹⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4617>

¹⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4616>

¹⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4615>

¹⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4614>

¹⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4613>

¹⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4611>

¹⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4610>

¹⁷⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4609>

¹⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4608>

¹⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4607>

- PR #4606¹⁷⁵⁵ - Local runtime module etc.
- PR #4604¹⁷⁵⁶ - Add config_registry module
- PR #4603¹⁷⁵⁷ - Deal with distributed modules in their respective CMakeLists.txt
- PR #4602¹⁷⁵⁸ - Small module fixes
- PR #4598¹⁷⁵⁹ - Making sure current_executor and service_executor functions are linked into the core library
- PR #4597¹⁷⁶⁰ - Adding broadcast_to/broadcast_from to collectives module
- PR #4596¹⁷⁶¹ - Fix performance regression in block_executor
- PR #4595¹⁷⁶² - Making sure main.cpp is built as a library if HPX_WITH_DYNAMIC_MAIN=OFF
- PR #4592¹⁷⁶³ - Futures module
- PR #4591¹⁷⁶⁴ - Adapting co_await support for C++20
- PR #4590¹⁷⁶⁵ - Adding missing exception test for for_loop()
- PR #4587¹⁷⁶⁶ - Move traits headers to hpx/modulename/traits directory
- PR #4586¹⁷⁶⁷ - Remove Travis CI config
- PR #4585¹⁷⁶⁸ - Update macOS test blacklist
- PR #4584¹⁷⁶⁹ - Attempting to fix missing symbols in stack trace
- PR #4583¹⁷⁷⁰ - Fixing bad static_cast
- PR #4582¹⁷⁷¹ - Changing download url for Windows prerequisites to circumvent bandwidth limitations
- PR #4581¹⁷⁷² - Adding missing using placeholder::_X
- PR #4579¹⁷⁷³ - Move get_stack_size_name and related functions
- PR #4575¹⁷⁷⁴ - Excluding unconditional definition of class backtrace from global header
- PR #4574¹⁷⁷⁵ - Changing return type of hardware_concurrency() to unsigned int
- PR #4570¹⁷⁷⁶ - Move tests to modules
- PR #4564¹⁷⁷⁷ - Reshuffle internal targets and add HPX::hpx_no_wrap_main target

¹⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4606>

¹⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4604>

¹⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4603>

¹⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4602>

¹⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4598>

¹⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4597>

¹⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4596>

¹⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4595>

¹⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4592>

¹⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4591>

¹⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4590>

¹⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4587>

¹⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4586>

¹⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4585>

¹⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4584>

¹⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4583>

¹⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4582>

¹⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4581>

¹⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4579>

¹⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4575>

¹⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4574>

¹⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4570>

¹⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4564>

- PR #4563¹⁷⁷⁸ - fix CMake option typo
- PR #4562¹⁷⁷⁹ - Unregister lock earlier to avoid holding it while suspending
- PR #4561¹⁷⁸⁰ - Adding test macros supporting custom output stream
- PR #4560¹⁷⁸¹ - Making sure hash_any::operator()() is linked into core library
- PR #4559¹⁷⁸² - Fixing compilation if HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION=On
- PR #4557¹⁷⁸³ - Improve spinlock implementation to perform better in high-contention situations
- PR #4553¹⁷⁸⁴ - Fix a runtime_ptr problem at shutdown when apex is enabled
- PR #4552¹⁷⁸⁵ - Add configuration option for making exceptions less noisy
- PR #4551¹⁷⁸⁶ - Clean up thread creation parameters
- PR #4549¹⁷⁸⁷ - Test FetchContent build on GitHub actions
- PR #4548¹⁷⁸⁸ - Fix stack size
- PR #4545¹⁷⁸⁹ - Fix header tests
- PR #4544¹⁷⁹⁰ - Fix a typo in sanitizer build
- PR #4541¹⁷⁹¹ - Add API to check if a thread pool exists
- PR #4540¹⁷⁹² - Making sure MPI support is enabled if MPI futures are used but networking is disabled
- PR #4538¹⁷⁹³ - Move channel documentation examples to examples directory
- PR #4536¹⁷⁹⁴ - Add generic allocator for execution policies
- PR #4534¹⁷⁹⁵ - Enable compatibility headers for thread_executors module
- PR #4532¹⁷⁹⁶ - Fixing broken url in README.rst
- PR #4531¹⁷⁹⁷ - Update scripts
- PR #4530¹⁷⁹⁸ - Make sure module API docs show up in correct order
- PR #4529¹⁷⁹⁹ - Adding missing template code to module creation script
- PR #4528¹⁸⁰⁰ - Make sure version module uses HPX's binary dir, not the parent's

¹⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4563>

¹⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4562>

¹⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4561>

¹⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4560>

¹⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4559>

¹⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4557>

¹⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4553>

¹⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4552>

¹⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4551>

¹⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4549>

¹⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4548>

¹⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4545>

¹⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4544>

¹⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4541>

¹⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4540>

¹⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4538>

¹⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4536>

¹⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4534>

1796 <https://github.com/STELLAR-GROUP/hpx/pull/4532>1797 <https://github.com/STELLAR-GROUP/hpx/pull/4531>1798 <https://github.com/STELLAR-GROUP/hpx/pull/4530>1799 <https://github.com/STELLAR-GROUP/hpx/pull/4529>1800 <https://github.com/STELLAR-GROUP/hpx/pull/4528>

- PR #4527¹⁸⁰¹ - Creating actions_base and actions module
- PR #4526¹⁸⁰² - Shared state for cv
- PR #4525¹⁸⁰³ - Changing sub-name sequencing for experimental namespace
- PR #4524¹⁸⁰⁴ - Add API guarantee notes to API reference documentation
- PR #4522¹⁸⁰⁵ - Enable and fix deprecation warnings in execution module
- PR #4521¹⁸⁰⁶ - Moves more miscellaneous files to modules
- PR #4520¹⁸⁰⁷ - Skip execution customization points when executor is known
- PR #4518¹⁸⁰⁸ - Module distributed lcos
- PR #4516¹⁸⁰⁹ - Fix various builds
- PR #4515¹⁸¹⁰ - Replace backslashes by slashes in windows paths
- PR #4514¹⁸¹¹ - Adding polymorphic_executor
- PR #4512¹⁸¹² - Adding C++20 jthread and stop_token
- PR #4510¹⁸¹³ - Attempt to fix APEX linking in external packages again
- PR #4508¹⁸¹⁴ - Only test pull requests (not all branches) with GitHub actions
- PR #4505¹⁸¹⁵ - Fix duplicate linking in tests (ODR violations)
- PR #4504¹⁸¹⁶ - Fix C++ standard handling
- PR #4503¹⁸¹⁷ - Add CMakeLists file check
- PR #4500¹⁸¹⁸ - Fix .clang-format version requirement comment
- PR #4499¹⁸¹⁹ - Attempting to fix hpx_init linking on macOS
- PR #4498¹⁸²⁰ - Fix compatibility of *pool_executor*
- PR #4496¹⁸²¹ - Removing superfluous SPDX tags
- PR #4494¹⁸²² - Module executors
- PR #4493¹⁸²³ - Pack traversal module

¹⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4527>

¹⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4526>

¹⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4525>

¹⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4524>

¹⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4522>

¹⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4521>

¹⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4520>

¹⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4518>

¹⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4516>

¹⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4515>

¹⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4514>

¹⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/4512>

¹⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4510>

¹⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4508>

¹⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4505>

¹⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4504>

¹⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4503>

¹⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4500>

¹⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4499>

¹⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4498>

¹⁸²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4496>

¹⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/4494>

¹⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/4493>

- PR #4492¹⁸²⁴ - Update copyright year in documentation
- PR #4491¹⁸²⁵ - Add missing current_executor header
- PR #4490¹⁸²⁶ - Update GitHub actions configs
- PR #4487¹⁸²⁷ - Properly dispatch exceptions thrown from hpx_main to be rethrown from hpx::init/hpx::stop
- PR #4486¹⁸²⁸ - Fixing an initialization order problem
- PR #4485¹⁸²⁹ - Move miscellaneous files to their rightful modules
- PR #4483¹⁸³⁰ - Clean up imported CMake target naming
- PR #4481¹⁸³¹ - Add vcpkg installation instructions
- PR #4479¹⁸³² - Add hints to allow to specify MIMALLOC_ROOT
- PR #4478¹⁸³³ - Async modules
- PR #4475¹⁸³⁴ - Fix rp init changes
- PR #4474¹⁸³⁵ - Use #pragma once in headers
- PR #4472¹⁸³⁶ - Add more descriptive error message when using x86 coroutines on non-x86 platforms
- PR #4467¹⁸³⁷ - Add mimalloc find cmake script
- PR #4465¹⁸³⁸ - Add thread_executors module
- PR #4464¹⁸³⁹ - Include module
- PR #4462¹⁸⁴⁰ - Merge hpx_init and hpx_wrap into one static library
- PR #4461¹⁸⁴¹ - Making thread_data test more realistic
- PR #4460¹⁸⁴² - Suppress MPI warnings in version.cpp
- PR #4459¹⁸⁴³ - Make sure pkgconfig applications link with hpx_init
- PR #4458¹⁸⁴⁴ - Added example demonstrating how to create and use a wrapping executor
- PR #4457¹⁸⁴⁵ - Fixing execution of thread exit functions
- PR #4456¹⁸⁴⁶ - Move backtrace files to debugging module

¹⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4492>

¹⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4491>

¹⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4490>

¹⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4487>

¹⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4486>

¹⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4485>

¹⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4483>

¹⁸³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4481>

¹⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/4479>

¹⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/4478>

¹⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4475>

¹⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4474>

¹⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4472>

¹⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4467>

¹⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4465>

¹⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4464>

¹⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4462>

¹⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4461>

¹⁸⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4460>

¹⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4459>

¹⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4458>

¹⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4457>

¹⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4456>

- PR #4455¹⁸⁴⁷ - Move deadlock_detection and maintain_queue_wait_times source files into schedulers module
- PR #4450¹⁸⁴⁸ - Fixing compilation with std::filesystem enabled
- PR #4449¹⁸⁴⁹ - Fixing build system to actually build variant test
- PR #4447¹⁸⁵⁰ - This fixes an obsolete #include
- PR #4446¹⁸⁵¹ - Resume tasks where they were suspended
- PR #4444¹⁸⁵² - Minor CUDA fixes
- PR #4443¹⁸⁵³ - Add missing tests to CircleCI config
- PR #4442¹⁸⁵⁴ - Adding a tag to all auto-generated files allowing for tools to visually distinguish those
- PR #4441¹⁸⁵⁵ - Adding performance counter type information
- PR #4440¹⁸⁵⁶ - Fixing MSVC build
- PR #4439¹⁸⁵⁷ - Link HPX::plugin and component privately in hpx_setup_target
- PR #4437¹⁸⁵⁸ - Adding a test that verifies the problem can be solved using a trait specialization
- PR #4434¹⁸⁵⁹ - Clean up Boost dependencies and copy string algorithms to new module
- PR #4433¹⁸⁶⁰ - Fixing compilation issues (!) if MPI parcelport is enabled
- PR #4431¹⁸⁶¹ - Ignore warnings about name mangling changing
- PR #4430¹⁸⁶² - Add performance_counters module
- PR #4428¹⁸⁶³ - Don't add compatibility headers to module API reference
- PR #4426¹⁸⁶⁴ - Add currently failing tests on GitHub actions to blacklist
- PR #4425¹⁸⁶⁵ - Clean up and correct minimum required versions
- PR #4424¹⁸⁶⁶ - Making sure hpx.lock_detection=0 works as advertised
- PR #4421¹⁸⁶⁷ - Making sure interval time stops underlying timer thread on termination
- PR #4417¹⁸⁶⁸ - Adding serialization support for std::variant (if available) and std::tuple
- PR #4415¹⁸⁶⁹ - Partially reverting changes applied by PR 4373

¹⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4455>

¹⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4450>

¹⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4449>

¹⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4447>

¹⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4446>

¹⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4444>

¹⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4443>

¹⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4442>

¹⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4441>

1856 <https://github.com/STELLAR-GROUP/hpx/pull/4440>1857 <https://github.com/STELLAR-GROUP/hpx/pull/4439>1858 <https://github.com/STELLAR-GROUP/hpx/pull/4437>1859 <https://github.com/STELLAR-GROUP/hpx/pull/4434>1860 <https://github.com/STELLAR-GROUP/hpx/pull/4433>1861 <https://github.com/STELLAR-GROUP/hpx/pull/4431>1862 <https://github.com/STELLAR-GROUP/hpx/pull/4430>1863 <https://github.com/STELLAR-GROUP/hpx/pull/4428>1864 <https://github.com/STELLAR-GROUP/hpx/pull/4426>1865 <https://github.com/STELLAR-GROUP/hpx/pull/4425>1866 <https://github.com/STELLAR-GROUP/hpx/pull/4424>1867 <https://github.com/STELLAR-GROUP/hpx/pull/4421>1868 <https://github.com/STELLAR-GROUP/hpx/pull/4417>1869 <https://github.com/STELLAR-GROUP/hpx/pull/4415>

- PR #4414¹⁸⁷⁰ - Added documentation for the compiler-wrapper script hpxcxx.in in creating_hpx_projects.rst
- PR #4413¹⁸⁷¹ - Merging from V1.4.1 release
- PR #4412¹⁸⁷² - Making sure to issue a warning if a file specified using –hpx:options-file is not found
- PR #4411¹⁸⁷³ - Make test specific to HPX_WITH_SHARED_PRIORITY_SCHEDULER
- PR #4407¹⁸⁷⁴ - Adding minimal MPI executor
- PR #4405¹⁸⁷⁵ - Fix cross pool injection test, use default scheduler as fallback
- PR #4404¹⁸⁷⁶ - Fix a race condition and clean-up usage of scheduler mode
- PR #4399¹⁸⁷⁷ - Add more threading modules
- PR #4398¹⁸⁷⁸ - Add CODEOWNERS file
- PR #4395¹⁸⁷⁹ - Adding a parameter to auto_chunk_size allowing to control the amount of iterations to measure
- PR #4393¹⁸⁸⁰ - Use appropriate cache-line size defaults for different platforms
- PR #4391¹⁸⁸¹ - Fixing use of allocator for C++20
- PR #4390¹⁸⁸² - Making –hpx:help behavior consistent
- PR #4388¹⁸⁸³ - Change the resource partitioner initialization
- PR #4387¹⁸⁸⁴ - Fix roll_release.sh
- PR #4386¹⁸⁸⁵ - Add warning messages for using thread binding options on macOS
- PR #4385¹⁸⁸⁶ - Cuda futures
- PR #4384¹⁸⁸⁷ - Make enabling dynamic hpx_main on non-Linux systems a configuration error
- PR #4383¹⁸⁸⁸ - Use configure_file for HPXCacheVariables.cmake
- PR #4382¹⁸⁸⁹ - Update spellchecking whitelist and fix more typos
- PR #4380¹⁸⁹⁰ - Add a helper function to get a future from a cuda stream
- PR #4379¹⁸⁹¹ - Add Windows and macOS CI with GitHub actions
- PR #4378¹⁸⁹² - Change C++ standard handling

¹⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4414>

¹⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4413>

¹⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4412>

¹⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4411>

¹⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4407>

¹⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4405>

¹⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4404>

¹⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4399>

¹⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4398>

¹⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4395>

¹⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4393>

¹⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4391>

¹⁸⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4390>

¹⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4388>

¹⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4387>

¹⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4386>

¹⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4385>

¹⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4384>

¹⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4383>

¹⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4382>

¹⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4380>

¹⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4379>

¹⁸⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4378>

- PR #4377¹⁸⁹³ - Remove Python scripts
- PR #4374¹⁸⁹⁴ - Adding overload for `hpx::init/hpx::start` for use with resource partitioner
- PR #4373¹⁸⁹⁵ - Adding test that verifies for 4369 to be fixed
- PR #4372¹⁸⁹⁶ - Another attempt at fixing the integral mismatch and conversion warnings
- PR #4370¹⁸⁹⁷ - Doc updates quick start
- PR #4368¹⁸⁹⁸ - Add a whitelist of words for weird spelling suggestions
- PR #4366¹⁸⁹⁹ - Suppress or fix clang-tidy-9 warnings
- PR #4365¹⁹⁰⁰ - Removing more Boost dependencies
- PR #4363¹⁹⁰¹ - Update clang-format config file for version 9
- PR #4362¹⁹⁰² - Fix indices typo
- PR #4361¹⁹⁰³ - Boost cleanup
- PR #4360¹⁹⁰⁴ - Move plugins
- PR #4358¹⁹⁰⁵ - Doc updates; generating documentation. Will likely need heavy editing.
- PR #4356¹⁹⁰⁶ - Remove some minor unused and unnecessary Boost includes
- PR #4355¹⁹⁰⁷ - Fix spellcheck step in CircleCI config
- PR #4354¹⁹⁰⁸ - Lightweight utility to hold a pack as members
- PR #4352¹⁹⁰⁹ - Minor fixes to the C++ standard detection for MSVC
- PR #4351¹⁹¹⁰ - Move generated documentation to hpx-docs repo
- PR #4347¹⁹¹¹ - Add cmake policy - CMP0074
- PR #4346¹⁹¹² - Remove file committed by mistake
- PR #4342¹⁹¹³ - Remove HCC and SYCL options from CMakeLists.txt
- PR #4341¹⁹¹⁴ - Fix launch process test with APEX enabled
- PR #4340¹⁹¹⁵ - Testing Cirrus CI

¹⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4377>

¹⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4374>

¹⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4373>

¹⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4372>

¹⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4370>

¹⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4368>

¹⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4366>

¹⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4365>

¹⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4363>

¹⁹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4362>

¹⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4361>

¹⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4360>

¹⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4358>

¹⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4356>

¹⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4355>

¹⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4354>

¹⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4352>

¹⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4351>

¹⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4347>

¹⁹¹² <https://github.com/STELLAR-GROUP/hpx/pull/4346>

¹⁹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4342>

¹⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4341>

¹⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4340>

- PR #4339¹⁹¹⁶ - Post 1.4.0 updates
- PR #4338¹⁹¹⁷ - Spelling corrections and CircleCI spell check
- PR #4333¹⁹¹⁸ - Flatten bound callables
- PR #4332¹⁹¹⁹ - This is a collection of mostly minor (cleanup) fixes
- PR #4331¹⁹²⁰ - This adds the missing tests for `async_colocated` and `async_continue_colocated`
- PR #4330¹⁹²¹ - Remove HPX.Compute host `default_executor`
- PR #4328¹⁹²² - Generate global header for `basic_execution` module
- PR #4327¹⁹²³ - Use `INTERNAL_FLAGS` option for all examples and components
- PR #4326¹⁹²⁴ - Usage of temporary allocator in assignment operator of `compute::vector`
- PR #4325¹⁹²⁵ - Use `hpx::threads::get_cache_line_size` in `prefetching.hpp`
- PR #4324¹⁹²⁶ - Enable compatibility headers option for execution module
- PR #4316¹⁹²⁷ - Add clang format indentppdirectives
- PR #4313¹⁹²⁸ - Introduce `index_pack` alias to pack of `size_t`
- PR #4312¹⁹²⁹ - Fixing compatibility header for `pack.hpp`
- PR #4311¹⁹³⁰ - Dataflow annotations for APEX
- PR #4309¹⁹³¹ - Update `launching_and_configuring_hpx_applications.rst`
- PR #4306¹⁹³² - Fix schedule hint not being taken from executor
- PR #4305¹⁹³³ - Implementing `hpx::functional::tag_invoke`
- PR #4304¹⁹³⁴ - Improve pack support utilities
- PR #4303¹⁹³⁵ - Remove errors module dependency on datastructures
- PR #4301¹⁹³⁶ - Clean up thread executors
- PR #4294¹⁹³⁷ - Logging revamp
- PR #4292¹⁹³⁸ - Remove SPDX tag from Boost License file to allow for github to recognize it

¹⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4339>

¹⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4338>

¹⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4333>

¹⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4332>

¹⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4331>

¹⁹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4330>

¹⁹²² <https://github.com/STELLAR-GROUP/hpx/pull/4328>

¹⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/4327>

¹⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4326>

¹⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4325>

¹⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4324>

¹⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4316>

¹⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4313>

¹⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4312>

¹⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4311>

¹⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4309>

¹⁹³² <https://github.com/STELLAR-GROUP/hpx/pull/4306>

¹⁹³³ <https://github.com/STELLAR-GROUP/hpx/pull/4305>

¹⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4304>

¹⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4303>

¹⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4301>

¹⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4294>

¹⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4292>

- PR #4291¹⁹³⁹ - Add format support for std::tm
- PR #4290¹⁹⁴⁰ - Simplify compatible tuples check
- PR #4288¹⁹⁴¹ - A lightweight take on boost::lexical_cast
- PR #4287¹⁹⁴² - Forking boost::lexical_cast as a new module
- PR #4277¹⁹⁴³ - MPI_futures
- PR #4270¹⁹⁴⁴ - Refactor future implementation
- PR #4265¹⁹⁴⁵ - Threading module
- PR #4259¹⁹⁴⁶ - Module naming base
- PR #4251¹⁹⁴⁷ - Local workrequesting scheduler
- PR #4250¹⁹⁴⁸ - Inline execution of scoped tasks, if possible
- PR #4247¹⁹⁴⁹ - Add execution in module headers
- PR #4246¹⁹⁵⁰ - Expose CMake targets officially
- PR #4239¹⁹⁵¹ - Doc updates miscellaneous (partially completed during Google Season of Docs)
- PR #4233¹⁹⁵² - Remove project() from modules + fix CMAKE_SOURCE_DIR issue
- PR #4231¹⁹⁵³ - Module local lcos
- PR #4207¹⁹⁵⁴ - Command line handling module
- PR #4206¹⁹⁵⁵ - Runtime configuration module
- PR #4141¹⁹⁵⁶ - Doc updates examples local to remote (partially completed during Google Season of Docs)
- PR #4091¹⁹⁵⁷ - Split runtime into local and distributed parts
- PR #4017¹⁹⁵⁸ - Require C++14

¹⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4291>

¹⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4290>

¹⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4288>

¹⁹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4287>

¹⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4277>

¹⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4270>

¹⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4265>

¹⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4259>

¹⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4251>

¹⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4250>

¹⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4247>

¹⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4246>

¹⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4239>

¹⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4233>

¹⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4231>

¹⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4207>

¹⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4206>

¹⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4141>

¹⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4091>

¹⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4017>

2.10.9 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 #4320¹⁹⁵⁹ - HPX 1.4.0 does not compile with gcc 10
- Issue #4336¹⁹⁶⁰ - Building HPX 1.4.0 with IO Counters breaks (Windows)
- Issue #4334¹⁹⁶¹ - HPX Debug and RelWithDebinfo builds on Windows not installing .pdb files
- Issue #4322¹⁹⁶² - Undefine VT1 and VT2 after boost includes
- Issue #4314¹⁹⁶³ - Compile error on 1.4.0
- Issue #4307¹⁹⁶⁴ - ld: error: duplicate symbol: freebsd_environ

Closed pull requests

- PR #4376¹⁹⁶⁵ - Attempt to fix some test build errors on Windows
- PR #4357¹⁹⁶⁶ - Adding missing #includes to fix gcc V10 linker problems
- PR #4353¹⁹⁶⁷ - Skip MPI_Finalize if MPI_Init is not called from HPX
- PR #4343¹⁹⁶⁸ - Give a hard error if IO counters are enabled on non-Linux systems
- PR #4337¹⁹⁶⁹ - Installing pdb files on Windows
- PR #4335¹⁹⁷⁰ - Adding capability to buildsystem to use an installed version of HPX
- PR #4315¹⁹⁷¹ - Forcing exported symbols from composable_guard to be linked into core library
- PR #4310¹⁹⁷² - Remove environment handling from exception.cpp

¹⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4320>

¹⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4336>

¹⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/4334>

¹⁹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/4322>

¹⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/4314>

¹⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4307>

¹⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4376>

¹⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4357>

¹⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4353>

¹⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4343>

¹⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4337>

¹⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4335>

¹⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4315>

¹⁹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4310>

2.10.10 HPX V1.4.0 (January 15, 2020)

General changes

- We have added the collectives `all_to_all` and `all_reduce`.
- We have added APIs for resiliency, which allows replication and replay for failed tasks. See the [documentation](#) for more details.
- Components can now be checkpointed.
- Performance improvements to schedulers and coroutines. A significant change is the addition of stackless coroutines. These are to be used for tasks that do not need to be suspended and can reduce overheads noticeably in applications with short tasks. A stackless coroutine can be created with the new stack size `thread_stacksize_nostack`.
- We have added an implementation of `unique_any`, which is a non-copyable version of `any`.
- The `shared_priority_queue_scheduler` has been improved. It now has lower overheads than the default scheduler in many situations. Unlike the default scheduler it fully supports NUMA scheduling hints. Enable it with the command line option `--hpx:queuing=shared-priority`. This scheduler should still be considered experimental, but its use is encouraged in real applications to help us make it production ready.
- We have added the performance counters `background-receive-duration` and `background-receive-overhead` for inspecting the time and overhead spent on receiving parcels in the background.
- Compilation time has been further improved when `HPX_WITH_NETWORKING=OFF`.
- We no longer require compiled Boost dependencies in certain configurations. This requires at least Boost 1.70, compiling on x86 with GCC 9, clang (libc++) 9, or VS2019 in C++17 mode. The dependency on Boost.Filesystem can explicitly be turned on with `HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY=ON` (it is off by default if the standard library supports `std::filesystem`). Boost.ProgramOptions has been copied into the HPX repository. We have a compatibility layer for users who must explicitly use Boost.ProgramOptions instead of the ProgramOptions provided by HPX. To remove the dependency `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY` must be explicitly set to OFF. This option will be removed in a future release. We have also removed several other header-only dependencies on Boost.
- It is now possible to use the process affinity mask set by tools like `numactl` and various batch environments with the command line option `--hpx:use-process-mask`. Enabling this option implies `--hpx:ignore-batch-env`.
- It is now possible to create standalone thread pools without starting the runtime. See the `standalone_thread_pool_executor.cpp` test in the `execution` module for an example.
- Tasks annotated with `hpx::util::annotated_function` now have their correct name when using APEX to generate OTF2 files.
- Cloning of APEX was defective in previous releases (it required manual intervention to check out the correct tag or branch). This has been fixed.
- The option `HPX_WITH_MORE_THAN_64_THREADS` is now ignored and will be removed in a future release. The value is instead derived directly from `HPX_WITH_MAX_CPU_COUNT` option.
- We have deprecated compiling in C++11 mode. The next release will require a C++14 capable compiler.
- We have deprecated support for the Vc library. This option will be replaced with SIMD support from the standard library in a future release.

- We have significantly refactored our CMake setup. This is intended to be a non-breaking change and will allow for using HPX through CMake targets in the future.
- We have continued modularizing the HPX library. In the process we have rearranged many header files into module-specific directories. All moved headers have compatibility headers which forward from the old location to the new location, together with a deprecation warning. The compatibility headers will eventually be removed.
- We now enforce formatting with `clang-format` on the majority of our source files.
- We have added SPDX license tags to all files.
- Many bugfixes.

Breaking changes

- The `HPX_WITH_THREAD_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_UNWRAPPED_COMPATIBILITY` option and the associated compatibility layer has been removed.

Closed issues

- Issue #4282¹⁹⁷³ - Build Issues with Release on Windows
- Issue #4278¹⁹⁷⁴ - Build Issues with CMake 3.14.4
- Issue #4273¹⁹⁷⁵ - Clients of HPX 1.4.0-rc2 with APEX are not linked to libhpx-apex
- Issue #4269¹⁹⁷⁶ - Building HPX 1.4.0-rc2 with support for APEX fails
- Issue #4263¹⁹⁷⁷ - Compilation fail on latest master
- Issue #4232¹⁹⁷⁸ - Configure of HPX project using CMake FetchContent fails
- Issue #4223¹⁹⁷⁹ - “Re-using the main() function as the main HPX entry point” doesn’t work
- Issue #4220¹⁹⁸⁰ - HPX won’t compile - error building `resource_partitioner`
- Issue #4215¹⁹⁸¹ - HPX 1.4.0rc1 does not link on s390x
- Issue #4204¹⁹⁸² - Trouble compiling HPX with Intel compiler
- Issue #4199¹⁹⁸³ - Refactor APEX to eliminate circular dependency
- Issue #4187¹⁹⁸⁴ - HPX can’t build on OSX
- Issue #4185¹⁹⁸⁵ - Simple debug output for development

¹⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/4282>

¹⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4278>

¹⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4273>

¹⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4269>

¹⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4263>

¹⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4232>

¹⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4223>

¹⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4220>

¹⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/4215>

¹⁹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/4204>

¹⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/4199>

¹⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4187>

¹⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4185>

- Issue #4182¹⁹⁸⁶ - @HPX_CONF_PREFIX@ is the empty string
- Issue #4169¹⁹⁸⁷ - HPX won't build with APEX
- Issue #4163¹⁹⁸⁸ - Add back HPX_LIBRARIES and HPX_INCLUDE_DIRS
- Issue #4161¹⁹⁸⁹ - It should be possible to call `find_package(HPX)` multiple times
- Issue #4155¹⁹⁹⁰ - `get_self_id()` for stackless threads returns `invalid_thread_id`
- Issue #4151¹⁹⁹¹ - build error with MPI code
- Issue #4150¹⁹⁹² - hpx won't build on POWER9 with clang 8
- Issue #4148¹⁹⁹³ - `cacheline_data` delivers poor performance with C++17 compared to C++14
- Issue #4144¹⁹⁹⁴ - target general in HPX_LIBRARIES does not exist
- Issue #4134¹⁹⁹⁵ - CMake Error when `-DHPX_WITH_HPXMP=ON`
- Issue #4132¹⁹⁹⁶ - parallel fill leaves elements unfilled
- Issue #4123¹⁹⁹⁷ - PAPI performance counters are inaccessible
- Issue #4118¹⁹⁹⁸ - `static_chunk_size` is not obeyed in scan algorithms
- Issue #4115¹⁹⁹⁹ - dependency chaining error with APEX
- Issue #4107²⁰⁰⁰ - Initializing runtime without entry point function and command line arguments
- Issue #4105²⁰⁰¹ - Bug in `hpx:bind=numa-balanced`
- Issue #4101²⁰⁰² - Bound tasks
- Issue #4100²⁰⁰³ - Add SPDX identifier to all files
- Issue #4085²⁰⁰⁴ - `hpx_topology` library should depend on `hwloc`
- Issue #4067²⁰⁰⁵ - HPX fails to build on macOS
- Issue #4056²⁰⁰⁶ - Building without thread manager idle backoff fails
- Issue #4052²⁰⁰⁷ - Enforce `clang-format` style for modules
- Issue #4032²⁰⁰⁸ - Simple hello world fails to launch correctly

¹⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4182>

¹⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4169>

¹⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4163>

¹⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4161>

¹⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4155>

¹⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/4151>

¹⁹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/4150>

¹⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/4148>

¹⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4144>

¹⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4134>

¹⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4132>

¹⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4123>

¹⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4118>

¹⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4115>

²⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4107>

²⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/4105>

²⁰⁰² <https://github.com/STELLAR-GROUP/hpx/issues/4101>

²⁰⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/4100>

²⁰⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/4085>

²⁰⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/4067>

²⁰⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/4056>

²⁰⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/4052>

²⁰⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/4032>

- Issue #4030²⁰⁰⁹ - Allow threads to skip context switching
- Issue #4029²⁰¹⁰ - Add support for mimalloc
- Issue #4005²⁰¹¹ - Can't link HPX when APEX enabled
- Issue #4002²⁰¹² - Missing header for algorithm module
- Issue #3989²⁰¹³ - conversion from long to unsigned int requires a narrowing conversion on MSVC
- Issue #3958²⁰¹⁴ - /statistics/average@ perf counter can't be created
- Issue #3953²⁰¹⁵ - CMake errors from `HPX_AddPseudoDependencies`
- Issue #3941²⁰¹⁶ - CMake error for APEX install target
- Issue #3940²⁰¹⁷ - Convert pseudo-doxygen function documentation into actual doxygen documentation
- Issue #3935²⁰¹⁸ - HPX compiler match too strict?
- Issue #3929²⁰¹⁹ - Buildbot failures on latest HPX stable
- Issue #3912²⁰²⁰ - I recommend publishing a version that does not depend on the boost library
- Issue #3890²⁰²¹ - `hpx.ini` not working
- Issue #3883²⁰²² - cuda compilation fails because of `-faligned-new`
- Issue #3879²⁰²³ - HPX fails to configure with `-DHPX_WITH_TESTS=OFF`
- Issue #3871²⁰²⁴ - `dataflow` does not support void allocators
- Issue #3867²⁰²⁵ - Latest HTML docs placed in wrong directory on GitHub pages
- Issue #3866²⁰²⁶ - Make sure all tests use `HPX_TEST*` macros and not `HPX_ASSERT`
- Issue #3857²⁰²⁷ - CMake all-keyword or all-plain for `target_link_libraries`
- Issue #3856²⁰²⁸ - `hpx_setup_target` adds rogue flags
- Issue #3850²⁰²⁹ - HPX fails to build on POWER8 with Clang7
- Issue #3848²⁰³⁰ - Remove `lva` member from `thread_init_data`
- Issue #3838²⁰³¹ - `hpx::parallel::count/count_if` failing tests

²⁰⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/4030>

²⁰¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/4029>

²⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/4005>

²⁰¹² <https://github.com/STELLAR-GROUP/hpx/issues/4002>

²⁰¹³ <https://github.com/STELLAR-GROUP/hpx/issues/3989>

²⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3958>

²⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3953>

²⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3941>

²⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3940>

²⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3935>

²⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3929>

²⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3912>

²⁰²¹ <https://github.com/STELLAR-GROUP/hpx/issues/3890>

²⁰²² <https://github.com/STELLAR-GROUP/hpx/issues/3883>

²⁰²³ <https://github.com/STELLAR-GROUP/hpx/issues/3879>

²⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3871>

²⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3867>

²⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3866>

²⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3857>

²⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3856>

²⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3850>

²⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3848>

²⁰³¹ <https://github.com/STELLAR-GROUP/hpx/issues/3838>

- Issue #3651²⁰³² - `hpx::parallel::transform_reduce` with non const reference as lambda parameter
- Issue #3560²⁰³³ - Apex integration with HPX not working properly
- Issue #3322²⁰³⁴ - No warning when mixing debug/release builds

Closed pull requests

- PR #4300²⁰³⁵ - Checks for `MPI_Init` being called twice
- PR #4299²⁰³⁶ - Small CMake fixes
- PR #4298²⁰³⁷ - Remove extra call to annotate function that messes up traces
- PR #4296²⁰³⁸ - Fixing collectives locking problem
- PR #4295²⁰³⁹ - Do not check `LICENSE_1_0.txt` for inspect violations
- PR #4293²⁰⁴⁰ - Applying two small changes fixing carious MSVC/Windows problems
- PR #4285²⁰⁴¹ - Delete `apex.hpp`
- PR #4276²⁰⁴² - Disable doxygen generation for `hpx/debugging/print.hpp` file
- PR #4275²⁰⁴³ - Make sure APEX is linked to even when not explicitly referenced
- PR #4272²⁰⁴⁴ - Fix pushing of documentation
- PR #4271²⁰⁴⁵ - Updating APEX tag, don't create new task_wrapper on `operator=` of `hpx_thread` object
- PR #4268²⁰⁴⁶ - Testing for noexcept function specializations in C++11/14 mode
- PR #4267²⁰⁴⁷ - Fixing MSVC warning
- PR #4266²⁰⁴⁸ - Make sure macOS Travis CI fails if build step fails
- PR #4264²⁰⁴⁹ - Clean up compatibility header options
- PR #4262²⁰⁵⁰ - Cleanup modules `CMakeLists.txt`
- PR #4261²⁰⁵¹ - Fixing HPX/APEX linking and dependencies for external projects like Phylanx
- PR #4260²⁰⁵² - Fix docs compilation problems
- PR #4258²⁰⁵³ - Couple of minor changes

²⁰³² <https://github.com/STELLAR-GROUP/hpx/issues/3651>

²⁰³³ <https://github.com/STELLAR-GROUP/hpx/issues/3560>

²⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3322>

²⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4300>

²⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4299>

²⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4298>

²⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4296>

²⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4295>

²⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4293>

²⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4285>

²⁰⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4276>

²⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4275>

²⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4272>

²⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4271>

²⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4268>

²⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4267>

²⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4266>

²⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4264>

²⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4262>

²⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4261>

²⁰⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4260>

²⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4258>

- PR #4257²⁰⁵⁴ - Fix apex annotation for async dispatch
- PR #4256²⁰⁵⁵ - Remove lambdas from assert expressions
- PR #4255²⁰⁵⁶ - Ignoring lock in all_to_all and all_reduce
- PR #4254²⁰⁵⁷ - Adding action specializations for noexcept functions
- PR #4253²⁰⁵⁸ - Move partlit.hpp to affinity module
- PR #4252²⁰⁵⁹ - Make mismatching build types a hard error in CMake
- PR #4249²⁰⁶⁰ - Scheduler improvement
- PR #4248²⁰⁶¹ - update hpxmp tag to v0.3.0
- PR #4245²⁰⁶² - Adding high performance channels
- PR #4244²⁰⁶³ - Ignore lock in ignore_while_locked_1485 test
- PR #4243²⁰⁶⁴ - Fix PAPI command line option documentation
- PR #4242²⁰⁶⁵ - Ignore lock in target_distribution_policy
- PR #4241²⁰⁶⁶ - Fix start_stop_callbacks test
- PR #4240²⁰⁶⁷ - Mostly fix clang CUDA compilation
- PR #4238²⁰⁶⁸ - Google Season of Docs updates to documentation; grammar edits.
- PR #4237²⁰⁶⁹ - fixing annotated task to use the name, not the desc
- PR #4236²⁰⁷⁰ - Move module print summary to modules
- PR #4235²⁰⁷¹ - Don't use alignas in cache_{aligned,line}_data
- PR #4234²⁰⁷² - Add basic overview sentence to all modules
- PR #4230²⁰⁷³ - Add OS X builds to Travis CI
- PR #4229²⁰⁷⁴ - Remove leftover queue compatibility checks
- PR #4226²⁰⁷⁵ - Fixing APEX shutdown by explicitly shutting down throttling
- PR #4225²⁰⁷⁶ - Allow CMAKE_INSTALL_PREFIX to be a relative path

²⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4257>

²⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4256>

²⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4255>

²⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4254>

²⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4253>

²⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4252>

2060 <https://github.com/STELLAR-GROUP/hpx/pull/4249>2061 <https://github.com/STELLAR-GROUP/hpx/pull/4248>2062 <https://github.com/STELLAR-GROUP/hpx/pull/4245>2063 <https://github.com/STELLAR-GROUP/hpx/pull/4244>2064 <https://github.com/STELLAR-GROUP/hpx/pull/4243>2065 <https://github.com/STELLAR-GROUP/hpx/pull/4242>2066 <https://github.com/STELLAR-GROUP/hpx/pull/4241>2067 <https://github.com/STELLAR-GROUP/hpx/pull/4240>2068 <https://github.com/STELLAR-GROUP/hpx/pull/4238>2069 <https://github.com/STELLAR-GROUP/hpx/pull/4237>2070 <https://github.com/STELLAR-GROUP/hpx/pull/4236>2071 <https://github.com/STELLAR-GROUP/hpx/pull/4235>2072 <https://github.com/STELLAR-GROUP/hpx/pull/4234>2073 <https://github.com/STELLAR-GROUP/hpx/pull/4230>2074 <https://github.com/STELLAR-GROUP/hpx/pull/4229>2075 <https://github.com/STELLAR-GROUP/hpx/pull/4226>2076 <https://github.com/STELLAR-GROUP/hpx/pull/4225>

- PR #4224²⁰⁷⁷ - Deprecate verbs parcelport
- PR #4222²⁰⁷⁸ - Update register_{thread,work} namespaces
- PR #4221²⁰⁷⁹ - Changing HPX_GCC_VERSION check from 70000 to 70300
- PR #4218²⁰⁸⁰ - Google Season of Docs updates to documentation; grammar edits.
- PR #4217²⁰⁸¹ - Google Season of Docs updates to documentation; grammar edits.
- PR #4216²⁰⁸² - Fixing gcc warning on 32bit platforms (integer truncation)
- PR #4214²⁰⁸³ - Apex callback refactoring
- PR #4213²⁰⁸⁴ - Clean up allocator checks for dependent projects
- PR #4212²⁰⁸⁵ - Google Season of Docs updates to documentation; grammar edits.
- PR #4211²⁰⁸⁶ - Google Season of Docs updates to documentation; contributing to hpx
- PR #4210²⁰⁸⁷ - Attempting to fix Intel compilation
- PR #4209²⁰⁸⁸ - Fix CUDA 10 build
- PR #4205²⁰⁸⁹ - Making sure that differences in CMAKE_BUILD_TYPE are not reported on multi-configuration cmake generators
- PR #4203²⁰⁹⁰ - Deprecate Vc
- PR #4202²⁰⁹¹ - Fix CUDA configuration
- PR #4200²⁰⁹² - Making sure hpx_wrap is not passed on to linker on non-Linux systems
- PR #4198²⁰⁹³ - Fix execution_agent.cpp compilation with GCC 5
- PR #4197²⁰⁹⁴ - Remove deprecated options for 1.4.0 release
- PR #4196²⁰⁹⁵ - minor fixes for building on OSX Darwin
- PR #4195²⁰⁹⁶ - Use full clone on CircleCI for pushing stable tag
- PR #4193²⁰⁹⁷ - Add scheduling hints to hello_world_distributed
- PR #4192²⁰⁹⁸ - Set up CUDA in HPXConfig.cmake
- PR #4191²⁰⁹⁹ - Export allocators root variables

²⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4224>

²⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4222>

²⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4221>

²⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4218>

²⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/4217>

²⁰⁸² <https://github.com/STELLAR-GROUP/hpx/pull/4216>

²⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/4214>

²⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4213>

²⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4212>

²⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4211>

²⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4210>

²⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4209>

²⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4205>

²⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4203>

²⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4202>

²⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4200>

²⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4198>

²⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4197>

²⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4196>

²⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4195>

²⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4193>

²⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4192>

²⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4191>

- PR #4190²¹⁰⁰ - Don't use `constexpr` in `thread_data` with GCC <= 6
- PR #4189²¹⁰¹ - Only use `quick_exit` if available
- PR #4188²¹⁰² - Google Season of Docs updates to documentation; writing single node hpx applications
- PR #4186²¹⁰³ - correct vc to cuda in cuda cmake
- PR #4184²¹⁰⁴ - Resetting some cached variables to make sure those are re-filled
- PR #4183²¹⁰⁵ - Fix `hpxcxx` configuration
- PR #4181²¹⁰⁶ - Rename base libraries var
- PR #4180²¹⁰⁷ - Move header left behind earlier to plugin module
- PR #4179²¹⁰⁸ - Moving `zip_iterator` and `transform_iterator` to `iterator_support` module
- PR #4178²¹⁰⁹ - Move checkpointing support to its own module
- PR #4177²¹¹⁰ - Small const fix to `basic_execution` module
- PR #4176²¹¹¹ - Add back `HPX_LIBRARIES` and friends to `HPXConfig.cmake`
- PR #4175²¹¹² - Make Vc public and add it to `HPXConfig.cmake`
- PR #4173²¹¹³ - Wait for runtime to be running before returning from `hpx::start`
- PR #4172²¹¹⁴ - More protection against shutdown problems in error handling scenarios.
- PR #4171²¹¹⁵ - Ignore lock in `condition_variable::wait`
- PR #4170²¹¹⁶ - Adding APEX dependency to MPI parcelport
- PR #4168²¹¹⁷ - Adding utility include
- PR #4167²¹¹⁸ - Add a condition to setup the external libraries
- PR #4166²¹¹⁹ - Add an `INTERNAL_FLAGS` option to link to `hpx_internal_flags`
- PR #4165²¹²⁰ - Forward `HPX_*` cmake cache variables to external projects
- PR #4164²¹²¹ - Affinity and batch environment modules
- PR #4162²¹²² - Handle `quick_exit`

²¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4190>

²¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4189>

²¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4188>

²¹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4186>

²¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4184>

²¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4183>

²¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4181>

²¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4180>

²¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4179>

²¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4178>

²¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4177>

²¹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4176>

²¹¹² <https://github.com/STELLAR-GROUP/hpx/pull/4175>

²¹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4173>

²¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4172>

²¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4171>

²¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4170>

²¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4168>

²¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4167>

²¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4166>

²¹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4165>

²¹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4164>

²¹²² <https://github.com/STELLAR-GROUP/hpx/pull/4162>

- PR #4160²¹²³ - Using `target_link_libraries` for cmake versions >= 3.12
- PR #4159²¹²⁴ - Make sure `HPX_WITH_NATIVE_TLS` is forwarded to dependent projects
- PR #4158²¹²⁵ - Adding allocator imported target as a dependency of allocator module
- PR #4157²¹²⁶ - Add `hpx_memory` as a dependency of parcelport plugins
- PR #4156²¹²⁷ - Stackless coroutines now can refer to themselves (through `get_self()` and friends)
- PR #4154²¹²⁸ - Added CMake policy CMP0060 for HPX applications.
- PR #4153²¹²⁹ - add header `iomanip` to tests and tool
- PR #4152²¹³⁰ - Casting MPI tag value
- PR #4149²¹³¹ - Add back private `m_desc` member variable in `program_options` module
- PR #4147²¹³² - Resource partitioner and threadmanager modules
- PR #4146²¹³³ - Google Season of Docs updates to documentation; creating hpx projects
- PR #4145²¹³⁴ - Adding basic support for stackless threads
- PR #4143²¹³⁵ - Exclude `test_client_1950` from all target
- PR #4142²¹³⁶ - Add a new `thread_pool_executor`
- PR #4140²¹³⁷ - Google Season of Docs updates to documentation; why hpx
- PR #4139²¹³⁸ - Remove runtime includes from coroutines module
- PR #4138²¹³⁹ - Forking `boost::intrusive_ptr` and adding it as `hpx::intrusive_ptr`
- PR #4137²¹⁴⁰ - Fixing TSS destruction
- PR #4136²¹⁴¹ - HPX.Compute modules
- PR #4133²¹⁴² - Fix `block_executor`
- PR #4131²¹⁴³ - Applying fixes based on reports from PVS Studio
- PR #4130²¹⁴⁴ - Adding missing header to build system
- PR #4129²¹⁴⁵ - Fixing compilation if `HPX_WITH_DATAPAR_VC` is enabled

²¹²³ <https://github.com/STELLAR-GROUP/hpx/pull/4160>

²¹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4159>

²¹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4158>

²¹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4157>

²¹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4156>

²¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4154>

²¹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4153>

²¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4152>

²¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4149>

²¹³² <https://github.com/STELLAR-GROUP/hpx/pull/4147>

²¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/4146>

²¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4145>

²¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4143>

²¹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4142>

²¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4140>

²¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4139>

²¹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4138>

²¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4137>

²¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/4136>

²¹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/4133>

²¹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/4131>

²¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4130>

²¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4129>

- PR #4128²¹⁴⁶ - Renaming `moveonly_any` to `unique_any`
- PR #4126²¹⁴⁷ - Attempt to fix `basic_any` constructor for gcc 7
- PR #4125²¹⁴⁸ - Changing `extra_archive_data` implementation
- PR #4124²¹⁴⁹ - Don't link to Boost.System unless required
- PR #4122²¹⁵⁰ - Add kernel launch helper utility (+saxy demo) and merge in octotiger changes
- PR #4121²¹⁵¹ - Fixing migration test if networking is disabled.
- PR #4120²¹⁵² - Google Season of Docs updates to documentation; hpx build system v1
- PR #4119²¹⁵³ - Making sure `chunk_size` and `max_chunk` are actually applied to parallel algorithms if specified
- PR #4117²¹⁵⁴ - Make CircleCI formatting check store diff
- PR #4116²¹⁵⁵ - Fix automatically setting C++ standard
- PR #4114²¹⁵⁶ - Module serialization
- PR #4113²¹⁵⁷ - Module datastructures
- PR #4111²¹⁵⁸ - Fixing performance regression introduced earlier
- PR #4110²¹⁵⁹ - Adding missing SPDX tags
- PR #4109²¹⁶⁰ - Overload for start without entry point/argv.
- PR #4108²¹⁶¹ - Making sure C++ standard is properly detected and propagated
- PR #4106²¹⁶² - use `std::round` for guaranteed rounding without errors
- PR #4104²¹⁶³ - Extend `scheduler_mode` with new `work_stealing` and task assignment modes
- PR #4103²¹⁶⁴ - Add this to lambda capture list
- PR #4102²¹⁶⁵ - Add SPDX license and check
- PR #4099²¹⁶⁶ - Module coroutines
- PR #4098²¹⁶⁷ - Fix append module path in module CMakeLists template
- PR #4097²¹⁶⁸ - Function tests

²¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4128>

²¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4126>

²¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4125>

²¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4124>

²¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4122>

²¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/4121>

²¹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/4120>

²¹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/4119>

²¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4117>

²¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4116>

²¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4114>

²¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4113>

²¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4111>

²¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4110>

²¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4109>

²¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/4108>

²¹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/4106>

²¹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/4104>

²¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4103>

²¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4102>

²¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4099>

²¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4098>

²¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4097>

- PR #4096²¹⁶⁹ - Removing return of `thread_result_type` from functions not needing them
- PR #4095²¹⁷⁰ - Stop-gap measure until cmake overhaul is in place
- PR #4094²¹⁷¹ - Deprecate `HPX_WITH_MORE_THAN_64_THREADS`
- PR #4093²¹⁷² - Fix initialization of `global_num_tasks` in `parallel_executor`
- PR #4092²¹⁷³ - Add support for mi-malloc
- PR #4090²¹⁷⁴ - Execution context
- PR #4089²¹⁷⁵ - Make counters in coroutines optional
- PR #4087²¹⁷⁶ - Making `hpx::util::any` compatible with C++17
- PR #4084²¹⁷⁷ - Making sure destination array for `std::transform` is properly resized
- PR #4083²¹⁷⁸ - Adapting `thread_queue_mc` to behave even if no 128bit atomics are available
- PR #4082²¹⁷⁹ - Fix compilation on GCC 5
- PR #4081²¹⁸⁰ - Adding option allowing to force using Boost.FileSystem
- PR #4080²¹⁸¹ - Updating module dependencies
- PR #4079²¹⁸² - Add missing tests for iterator_support module
- PR #4078²¹⁸³ - Disable parcel-layer if networking is disabled
- PR #4077²¹⁸⁴ - Add missing include that causes build fails
- PR #4076²¹⁸⁵ - Enable compatibility headers for functional module
- PR #4075²¹⁸⁶ - Coroutines module
- PR #4073²¹⁸⁷ - Use `configure_file` for generated files in modules
- PR #4071²¹⁸⁸ - Fixing MPI detection for PMIx
- PR #4070²¹⁸⁹ - Fix macOS builds
- PR #4069²¹⁹⁰ - Moving more facilities to the collectives module
- PR #4068²¹⁹¹ - Adding main HPX `#include` directory to modules

²¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4096>

²¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4095>

²¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/4094>

²¹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/4093>

²¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/4092>

²¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4090>

²¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4089>

²¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4087>

²¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4084>

2178 <https://github.com/STELLAR-GROUP/hpx/pull/4083>2179 <https://github.com/STELLAR-GROUP/hpx/pull/4082>2180 <https://github.com/STELLAR-GROUP/hpx/pull/4081>2181 <https://github.com/STELLAR-GROUP/hpx/pull/4080>2182 <https://github.com/STELLAR-GROUP/hpx/pull/4079>2183 <https://github.com/STELLAR-GROUP/hpx/pull/4078>2184 <https://github.com/STELLAR-GROUP/hpx/pull/4077>2185 <https://github.com/STELLAR-GROUP/hpx/pull/4076>2186 <https://github.com/STELLAR-GROUP/hpx/pull/4075>2187 <https://github.com/STELLAR-GROUP/hpx/pull/4073>2188 <https://github.com/STELLAR-GROUP/hpx/pull/4071>2189 <https://github.com/STELLAR-GROUP/hpx/pull/4070>2190 <https://github.com/STELLAR-GROUP/hpx/pull/4069>2191 <https://github.com/STELLAR-GROUP/hpx/pull/4068>

- PR #4066²¹⁹² - Switching the use of `message(STATUS "...")` to `hpx_info`
- PR #4065²¹⁹³ - Move Boost.Filesystem handling to filesystem module
- PR #4064²¹⁹⁴ - Fix program_options test with older boost versions
- PR #4062²¹⁹⁵ - The `cpu_features` tool fails to compile on anything but x86 architectures
- PR #4061²¹⁹⁶ - Add `clang-format` checking step for modules
- PR #4060²¹⁹⁷ - Making sure `HPX_IDLE_BACKOFF_TIME_MAX` is always defined (even if its unused)
- PR #4059²¹⁹⁸ - Renaming module `hpx_parallel_executors` into `hpx_execution`
- PR #4058²¹⁹⁹ - Do not build networking tests when networking disabled
- PR #4057²²⁰⁰ - Printing configuration summary for modules as well
- PR #4055²²⁰¹ - Google Season of Docs updates to documentation; hpx build systems
- PR #4054²²⁰² - Add troubleshooting section to manual
- PR #4051²²⁰³ - Add more variations to `future_overhead` test
- PR #4050²²⁰⁴ - Creating plugin module
- PR #4049²²⁰⁵ - Move missing modules tests
- PR #4047²²⁰⁶ - Add boost/filesystem headers to inspect deprecated headers
- PR #4045²²⁰⁷ - Module functional
- PR #4043²²⁰⁸ - Fix preconditions and error messages for suspension functions
- PR #4041²²⁰⁹ - Pass `HPX_STANDARD` on to dependent projects via `HPXConfig.cmake`
- PR #4040²²¹⁰ - Program options module
- PR #4039²²¹¹ - Moving non-serializable `any` (`any_nonser`) to datastructures module
- PR #4038²²¹² - Adding MPark's variant (V1.4.0) to HPX
- PR #4037²²¹³ - Adding resiliency module
- PR #4036²²¹⁴ - Add C++17 filesystem compatibility header

²¹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/4066>

²¹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/4065>

²¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4064>

²¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4062>

²¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4061>

²¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4060>

²¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4059>

²¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4058>

²²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4057>

²²⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/4055>

²²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/4054>

²²⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/4051>

²²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4050>

²²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4049>

²²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4047>

²²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4045>

²²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4043>

²²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4041>

²²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4040>

²²¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/4039>

²²¹² <https://github.com/STELLAR-GROUP/hpx/pull/4038>

²²¹³ <https://github.com/STELLAR-GROUP/hpx/pull/4037>

²²¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4036>

- PR #4035²²¹⁵ - Fixing support for mpirun
- PR #4028²²¹⁶ - CMake to target based directives
- PR #4027²²¹⁷ - Remove GitLab CI configuration
- PR #4026²²¹⁸ - Threading refactoring
- PR #4025²²¹⁹ - Refactoring thread queue configuration options
- PR #4024²²²⁰ - Fix padding calculation in `cache_aligned_data.hpp`
- PR #4023²²²¹ - Fixing Codacy issues
- PR #4022²²²² - Make sure process mask option is passed to `affinity_data`
- PR #4021²²²³ - Warn about compiling in C++11 mode
- PR #4020²²²⁴ - Module concurrency
- PR #4019²²²⁵ - Module topology
- PR #4018²²²⁶ - Update deprecated header in `thread_queue_mc.hpp`
- PR #4015²²²⁷ - Avoid overwriting artifacts
- PR #4014²²²⁸ - Future overheads
- PR #4013²²²⁹ - Update URL to test output conversion script
- PR #4012²²³⁰ - Fix CUDA compilation
- PR #4011²²³¹ - Fixing cyclic dependencies between modules
- PR #4010²²³² - Ignore stable tag on CircleCI
- PR #4009²²³³ - Check circular dependencies in a circle ci step
- PR #4008²²³⁴ - Extend cache aligned data to handle tuple-like data
- PR #4007²²³⁵ - Fixing migration for components that have actions returning a client
- PR #4006²²³⁶ - Move `is_value_proxy.hpp` to algorithms module
- PR #4004²²³⁷ - Shorten CTest timeout on CircleCI

²²¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4035>

²²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4028>

²²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4027>

²²¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4026>

²²¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4025>

²²²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4024>

²²²¹ <https://github.com/STELLAR-GROUP/hpx/pull/4023>

²²²² <https://github.com/STELLAR-GROUP/hpx/pull/4022>

²²²³ <https://github.com/STELLAR-GROUP/hpx/pull/4021>

²²²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4020>

²²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4019>

²²²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4018>

²²²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4015>

²²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4014>

²²²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4013>

²²³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4012>

²²³¹ <https://github.com/STELLAR-GROUP/hpx/pull/4011>

²²³² <https://github.com/STELLAR-GROUP/hpx/pull/4010>

²²³³ <https://github.com/STELLAR-GROUP/hpx/pull/4009>

²²³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/4008>

²²³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/4007>

²²³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/4006>

²²³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/4004>

- PR #4003²²³⁸ - Refactoring to remove (internal) dependencies
- PR #4001²²³⁹ - Exclude tests from all target
- PR #4000²²⁴⁰ - Module errors
- PR #3999²²⁴¹ - Enable support for compatibility headers for logging module
- PR #3998²²⁴² - Add process thread binding option
- PR #3997²²⁴³ - Export handle_assert function
- PR #3996²²⁴⁴ - Attempt to solve issue where -latomic does not support 128bit atomics
- PR #3993²²⁴⁵ - Make sure __LINE__ is an unsigned
- PR #3991²²⁴⁶ - Fix dependencies and flags for header tests
- PR #3990²²⁴⁷ - Documentation tags fixes
- PR #3988²²⁴⁸ - Adding missing solution folder for format module test
- PR #3987²²⁴⁹ - Move runtime-dependent functions out of command line handling
- PR #3986²²⁵⁰ - Fix CMake configuration with PAPI on
- PR #3985²²⁵¹ - Module timing
- PR #3984²²⁵² - Fix default behaviour of paths in add_hpx_component
- PR #3982²²⁵³ - Parallel executors module
- PR #3981²²⁵⁴ - Segmented algorithms module
- PR #3980²²⁵⁵ - Module logging
- PR #3979²²⁵⁶ - Module util
- PR #3978²²⁵⁷ - Fix clang-tidy step on CircleCI
- PR #3977²²⁵⁸ - Fixing solution folders for moved components
- PR #3976²²⁵⁹ - Module format
- PR #3975²²⁶⁰ - Enable deprecation warnings on CircleCI

²²³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/4003>

²²³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/4001>

²²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/4000>

²²⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3999>

²²⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3998>

²²⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3997>

²²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3996>

²²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3993>

²²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3991>

²²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3990>

²²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3988>

²²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3987>

²²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3986>

²²⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3985>

²²⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3984>

²²⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3982>

²²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3981>

²²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3980>

²²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3979>

²²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3978>

²²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3977>

²²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3976>

²²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3975>

- PR #3974²²⁶¹ - Fix typos in documentation
- PR #3973²²⁶² - Fix compilation with GCC 9
- PR #3972²²⁶³ - Add condition to clone apex + use of new cmake var APEX_ROOT
- PR #3971²²⁶⁴ - Add testing module
- PR #3968²²⁶⁵ - Remove unneeded file in hardware module
- PR #3967²²⁶⁶ - Remove leftover PIC settings from main CMakeLists.txt
- PR #3966²²⁶⁷ - Add missing export option in add_hpx_module
- PR #3965²²⁶⁸ - Change current_function_helper back to non-constexpr
- PR #3964²²⁶⁹ - Fixing merge problems
- PR #3962²²⁷⁰ - Add a trait for std::array for unwrapping
- PR #3961²²⁷¹ - Making hpx::util::tuple<Ts...> and std::tuple<Ts...> convertible
- PR #3960²²⁷² - fix compilation with CUDA 10 and GCC 6
- PR #3959²²⁷³ - Fix C++11 incompatibility
- PR #3957²²⁷⁴ - Algorithms module
- PR #3956²²⁷⁵ - [HPX_AddModule] Fix lower name var to upper
- PR #3955²²⁷⁶ - Fix CMake configuration with examples off and tests on
- PR #3954²²⁷⁷ - Move components to separate subdirectory in root of repository
- PR #3952²²⁷⁸ - Update papi.cpp
- PR #3951²²⁷⁹ - Exclude modules header tests from all target
- PR #3950²²⁸⁰ - Adding all_reduce facility to collectives module
- PR #3949²²⁸¹ - This adds a configuration file that will cause for stale issues to be automatically closed
- PR #3948²²⁸² - Fixing ALPS environment
- PR #3947²²⁸³ - Add major compiler version check for building hpx as a binary package

²²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3974>

²²⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3973>

²²⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3972>

²²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3971>

²²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3968>

²²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3967>

²²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3966>

²²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3965>

²²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3964>

²²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3962>

²²⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3961>

²²⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3960>

²²⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3959>

²²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3957>

²²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3956>

²²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3955>

²²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3954>

²²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3952>

²²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3951>

²²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3950>

²²⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3949>

²²⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3948>

²²⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3947>

- PR #3946²²⁸⁴ - [Modules] Move the location of the generated headers
- PR #3945²²⁸⁵ - Simplify tests and examples cmake
- PR #3943²²⁸⁶ - Remove example module
- PR #3942²²⁸⁷ - Add NOEXPORT option to `add_hpx_{component,library}`
- PR #3938²²⁸⁸ - Use https for CDash submissions
- PR #3937²²⁸⁹ - Add HPX_WITH_BUILD_BINARY_PACKAGE to the compiler check (refs #3935)
- PR #3936²²⁹⁰ - Fixing installation of binaries on windows
- PR #3934²²⁹¹ - Add set function for `sliding_semaphore max_difference`
- PR #3933²²⁹² - Remove cudadevrt from compile/link flags as it breaks downstream projects
- PR #3932²²⁹³ - Fixing 3929
- PR #3931²²⁹⁴ - Adding `all_to_all`
- PR #3930²²⁹⁵ - Add test demonstrating the use of broadcast with component actions
- PR #3928²²⁹⁶ - fixed number of tasks and number of threads for heterogeneous slurm environments
- PR #3927²²⁹⁷ - Moving Cache module's tests into separate solution folder
- PR #3926²²⁹⁸ - Move unit tests to cache module
- PR #3925²²⁹⁹ - Move version check to config module
- PR #3924²³⁰⁰ - Add schedule hint executor parameters
- PR #3923²³⁰¹ - Allow aligning objects bigger than the cache line size
- PR #3922²³⁰² - Add Windows builds with Travis CI
- PR #3921²³⁰³ - Add ccls cache directory to gitignore
- PR #3920²³⁰⁴ - Fix `git_external` fetching of tags
- PR #3905²³⁰⁵ - Correct rostambod url. Fix typo in doc
- PR #3904²³⁰⁶ - Fix bug in `context_base.hpp`

²²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3946>

²²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3945>

²²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3943>

²²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3942>

²²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3938>

²²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3937>

²²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3936>

²²⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3934>

²²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3933>

²²⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3932>

²²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3931>

²²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3930>

²²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3928>

²²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3927>

²²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3926>

²²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3925>

²³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3924>

²³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3923>

²³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3922>

²³⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3921>

²³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3920>

²³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3905>

²³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3904>

- PR #3903²³⁰⁷ - Adding new performance counters
- PR #3902²³⁰⁸ - Add add_hpx_module function
- PR #3901²³⁰⁹ - Factoring out container remapping into a separate trait
- PR #3900²³¹⁰ - Making sure errors during command line processing are properly reported and will not cause assertions
- PR #3899²³¹¹ - Remove old compatibility bases from make_action
- PR #3898²³¹² - Make parameter size be of type size_t
- PR #3897²³¹³ - Making sure all tests are disabled if HPX_WITH_TESTS=OFF
- PR #3895²³¹⁴ - Add documentation for annotated_function
- PR #3894²³¹⁵ - Working around VS2019 problem with make_action
- PR #3892²³¹⁶ - Avoid MSVC compatibility warning in internal allocator
- PR #3891²³¹⁷ - Removal of the default intel config include
- PR #3888²³¹⁸ - Fix async_customization dataflow example and Clarify what's being tested
- PR #3887²³¹⁹ - Add Doxygen documentation
- PR #3882²³²⁰ - Minor docs fixes
- PR #3880²³²¹ - Updating APEX version tag
- PR #3878²³²² - Making sure symbols are properly exported from modules (needed for Windows/MacOS)
- PR #3877²³²³ - Documentation
- PR #3876²³²⁴ - Module hardware
- PR #3875²³²⁵ - Converted typedefs in actions submodule to using directives
- PR #3874²³²⁶ - Allow one to suppress target keywords in hpx_setup_target for backwards compatibility
- PR #3873²³²⁷ - Add scripts to create releases and generate lists of PRs and issues
- PR #3872²³²⁸ - Fix latest HTML docs location
- PR #3870²³²⁹ - Module cache

²³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3903>

²³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3902>

²³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3901>

²³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3900>

²³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3899>

²³¹² <https://github.com/STELLAR-GROUP/hpx/pull/3898>

²³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3897>

²³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3895>

²³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3894>

²³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3892>

²³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3891>

²³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3888>

²³¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3887>

²³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3882>

²³²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3880>

²³²² <https://github.com/STELLAR-GROUP/hpx/pull/3878>

²³²³ <https://github.com/STELLAR-GROUP/hpx/pull/3877>

²³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3876>

²³²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3875>

²³²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3874>

²³²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3873>

²³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3872>

²³²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3870>

- PR #3869²³³⁰ - Post 1.3.0 version bumps
- PR #3868²³³¹ - Replace the macro HPX_ASSERT by HPX_TEST in tests
- PR #3845²³³² - Assertion module
- PR #3839²³³³ - Make tuple serialization non-intrusive
- PR #3832²³³⁴ - Config module
- PR #3799²³³⁵ - Remove compat namespace and its contents
- PR #3701²³³⁶ - MoodyCamel lockfree
- PR #3496²³³⁷ - Disabling MPI's (deprecated) C++ interface
- PR #3192²³³⁸ - Move type info into hpx::debug namespace and add print helper functions
- PR #3159²³³⁹ - Support Checkpointing Components

2.10.11 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.

²³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3869>

²³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3868>

²³³² <https://github.com/STELLAR-GROUP/hpx/pull/3845>

²³³³ <https://github.com/STELLAR-GROUP/hpx/pull/3839>

²³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3832>

²³³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3799>

²³³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3701>

²³³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3496>

²³³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3192>

²³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3159>

Breaking changes

- Executable and library targets are now created without the _exe and _lib suffix respectively. For example, the target 1d_stencil_1_exe is now simply called 1d_stencil_1.
- We have removed the following deprecated functionality: queue, scoped_unlock, and support for input iterators in algorithms.
- We have turned off the compatibility layer for unwrapped by default. The functionality will be removed in the next release. The option can still be turned on using the CMake²³⁴⁰ 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 #3863²³⁴¹ - shouldn't “-faligned-new” be a usage requirement?
- Issue #3841²³⁴² - Build error with msvc 19 caused by SFINAE and C++17
- Issue #3836²³⁴³ - master branch does not build with idle rate counters enabled
- Issue #3819²³⁴⁴ - Add debug suffix to modules built in debug mode
- Issue #3817²³⁴⁵ - HPX_INCLUDE_DIRS contains non-existent directory
- Issue #3810²³⁴⁶ - Source groups are not created for files in modules
- Issue #3805²³⁴⁷ - HPX won't compile with -DHpx_WITH_APEX=TRUE
- Issue #3792²³⁴⁸ - Barrier Hangs When Locality Zero not included
- Issue #3778²³⁴⁹ - Replace throw() with noexcept
- Issue #3763²³⁵⁰ - configurable sort limit per task
- Issue #3758²³⁵¹ - dataflow doesn't convert future<future<T>> to future<T>
- Issue #3757²³⁵² - When compiling undefined reference to hpx::hpx_check_version_1_2 HPX V1.2.1, Ubuntu 18.04.01 Server Edition
- Issue #3753²³⁵³ --hpx:list-counters=full crashes

²³⁴⁰ <https://www.cmake.org>

²³⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/3863>

²³⁴² <https://github.com/STELLAR-GROUP/hpx/issues/3841>

²³⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/3836>

²³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3819>

2345 <https://github.com/STELLAR-GROUP/hpx/issues/3817>2346 <https://github.com/STELLAR-GROUP/hpx/issues/3810>2347 <https://github.com/STELLAR-GROUP/hpx/issues/3805>2348 <https://github.com/STELLAR-GROUP/hpx/issues/3792>2349 <https://github.com/STELLAR-GROUP/hpx/issues/3778>2350 <https://github.com/STELLAR-GROUP/hpx/issues/3763>2351 <https://github.com/STELLAR-GROUP/hpx/issues/3758>2352 <https://github.com/STELLAR-GROUP/hpx/issues/3757>2353 <https://github.com/STELLAR-GROUP/hpx/issues/3753>

- Issue #3746²³⁵⁴ - Detection of MPI with pmix
- Issue #3744²³⁵⁵ - Separate spinlock from same cacheline as internal data for all LCOs
- Issue #3743²³⁵⁶ - hpxcxx's shebang doesn't specify the python version
- Issue #3738²³⁵⁷ - Unable to debug parcelport on a single node
- Issue #3735²³⁵⁸ - Latest master: Can't compile in MSVC
- Issue #3731²³⁵⁹ - `util::bound` seems broken on Clang with older libstdc++
- Issue #3724²³⁶⁰ - Allow to pre-set command line options through environment
- Issue #3723²³⁶¹ - examples/resource_partitioner build issue on master branch / ubuntu 18
- Issue #3721²³⁶² - faced a building error
- Issue #3720²³⁶³ - Hello World example fails to link
- Issue #3719²³⁶⁴ - pkg-config produces invalid output: -l-pthread
- Issue #3718²³⁶⁵ - Please make the python executable configurable through cmake
- Issue #3717²³⁶⁶ - interested to contribute to the organisation
- Issue #3699²³⁶⁷ - Remove 'HPX runtime' executable
- Issue #3698²³⁶⁸ - Ignore all locks while handling asserts
- Issue #3689²³⁶⁹ - Incorrect and inconsistent website structure <http://stellar.cct.lsu.edu/downloads/>.
- Issue #3681²³⁷⁰ - Broken links on <http://stellar.cct.lsu.edu/2015/05/hpx-archives-now-on-gmane/>
- Issue #3676²³⁷¹ - HPX master built from source, cmake fails to link main.cpp example in docs
- Issue #3673²³⁷² - HPX build fails with `std::atomic` missing error
- Issue #3670²³⁷³ - Generate PDF again from documentation (with Sphinx)
- Issue #3643²³⁷⁴ - Warnings when compiling HPX 1.2.1 with gcc 9
- Issue #3641²³⁷⁵ - Trouble with using ranges-v3 and `hpx::parallel::reduce`
- Issue #3639²³⁷⁶ - `util::unwrapping` does not work well with member functions

²³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3746>

²³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3744>

²³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3743>

²³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3738>

²³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3735>

²³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3731>

²³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3724>

²³⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/3723>

²³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/3721>

²³⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/3720>

²³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3719>

²³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3718>

²³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3717>

²³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3699>

²³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3698>

²³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3689>

²³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3681>

²³⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/3676>

²³⁷² <https://github.com/STELLAR-GROUP/hpx/issues/3673>

²³⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/3670>

²³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3643>

²³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3641>

²³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3639>

- Issue #3634²³⁷⁷ - The build fails if `shared_future<>::then` is called with a thread executor
- Issue #3622²³⁷⁸ - VTune Amplifier 2019 not working with `use_itt_notify=1`
- Issue #3616²³⁷⁹ - HPX Fails to Build with CUDA 10
- Issue #3612²³⁸⁰ - False sharing of scheduling counters
- Issue #3609²³⁸¹ - `executor_parameters timeout` with `gcc <= 7` and Debug mode
- Issue #3601²³⁸² - Misleading error message on power pc for `rdtsc` and `rdtscp`
- Issue #3598²³⁸³ - Build of some examples fails when using `Vc`
- Issue #3594²³⁸⁴ - Error: The number of OS threads requested (20) does not match the number of threads to bind (12): `HPX(bad_parameter)`
- Issue #3592²³⁸⁵ - Undefined Reference Error
- Issue #3589²³⁸⁶ - include could not find load file: `HPX_Utils.cmake`
- Issue #3587²³⁸⁷ - HPX won't compile on POWER8 with Clang 7
- Issue #3583²³⁸⁸ - Fedora and openSUSE instructions missing on "Distribution Packages" page
- Issue #3578²³⁸⁹ - Build error when configuring with `HPX_HAVE_ALGORITHM_INPUT_ITERATOR_SUPPORT=ON`
- Issue #3575²³⁹⁰ - Merge openSUSE reproducible patch
- Issue #3570²³⁹¹ - Update HPX to work with the latest VC version
- Issue #3567²³⁹² - Build succeed and make failed for `hpx:cout`
- Issue #3565²³⁹³ - Polymorphic simple component destructor not getting called
- Issue #3559²³⁹⁴ - 1.2.0 is missing from download page
- Issue #3554²³⁹⁵ - Clang 6.0 warning of hiding overloaded virtual function
- Issue #3510²³⁹⁶ - Build on ppc64 fails
- Issue #3482²³⁹⁷ - Improve error message when `HPX_WITH_MAX_CPU_COUNT` is too low for given system
- Issue #3453²³⁹⁸ - Two HPX applications can't run at the same time.
- Issue #3452²³⁹⁹ - Scaling issue on the change to 2 NUMA domains

²³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3634>

²³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3622>

²³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3616>

²³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3612>

²³⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/3609>

²³⁸² <https://github.com/STELLAR-GROUP/hpx/issues/3601>

²³⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/3598>

²³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3594>

²³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3592>

²³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3589>

²³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3587>

²³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3583>

²³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3578>

²³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3575>

²³⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/3570>

²³⁹² <https://github.com/STELLAR-GROUP/hpx/issues/3567>

²³⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/3565>

²³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3559>

²³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3554>

²³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3510>

²³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3482>

²³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3453>

²³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3452>

- Issue #3442²⁴⁰⁰ - HPX set_difference, set_intersection failure cases
- Issue #3437²⁴⁰¹ - Ensure parent_task pointer when child task is created and child/parent are on same locality
- Issue #3255²⁴⁰² - Suspension with lock for --hpx:list-component-types
- Issue #3034²⁴⁰³ - Use C++17 structured bindings for serialization
- Issue #2999²⁴⁰⁴ - Change thread scheduling use of `size_t` for thread indexing

Closed pull requests

- PR #3865²⁴⁰⁵ - adds `hpx_target_compile_option_if_available`
- PR #3864²⁴⁰⁶ - Helper functions that are useful in numa binding and testing of allocator
- PR #3862²⁴⁰⁷ - Temporary fix to `local_dataflow_boost_small_vector` test
- PR #3860²⁴⁰⁸ - Add cache line padding to intermediate results in for loop reduction
- PR #3859²⁴⁰⁹ - Remove `HPX_TLL_PUBLIC` and `HPX_TLL_PRIVATE` from CMake files
- PR #3858²⁴¹⁰ - Add compile flags and definitions to modules
- PR #3851²⁴¹¹ - update `hpxmp` release tag to v0.2.0
- PR #3849²⁴¹² - Correct `BOOST_ROOT` variable name in quick start guide
- PR #3847²⁴¹³ - Fix `attach_debugger` configuration option
- PR #3846²⁴¹⁴ - Add tests for `libs` header tests
- PR #3844²⁴¹⁵ - Fixing `source_groups` in preprocessor module to properly handle compatibility headers
- PR #3843²⁴¹⁶ - This fixes the `launch_process/launched_process` pair of tests
- PR #3842²⁴¹⁷ - Fix macro call with `ITTNOTIFY` enabled
- PR #3840²⁴¹⁸ - Fixing SLURM environment parsing
- PR #3837²⁴¹⁹ - Fixing misplaced `#endif`
- PR #3835²⁴²⁰ - make all latch members protected for consistency
- PR #3834²⁴²¹ - Disable `transpose_block_numa` example on CircleCI

²⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3442>

²⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/3437>

²⁴⁰² <https://github.com/STELLAR-GROUP/hpx/issues/3255>

²⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/3034>

²⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2999>

²⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3865>

²⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3864>

²⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3862>

²⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3860>

²⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3859>

²⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3858>

²⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3851>

²⁴¹² <https://github.com/STELLAR-GROUP/hpx/pull/3849>

²⁴¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3847>

²⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3846>

²⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3844>

²⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3843>

²⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3842>

²⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3840>

²⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3837>

²⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3835>

²⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3834>

- PR #3833²⁴²² - make latch **counter** protected for deriving latch in hpxmp
- PR #3831²⁴²³ - Fix CircleCI config for modules
- PR #3830²⁴²⁴ - minor fix: option HPX_WITH_TEST was not working correctly
- PR #3828²⁴²⁵ - Avoid for binaries that depend on HPX to directly link against internal modules
- PR #3827²⁴²⁶ - Adding shortcut for `hpx::get_ptr<>(sync, id)` for a local, non-migratable objects
- PR #3826²⁴²⁷ - Fix and update modules documentation
- PR #3825²⁴²⁸ - Updating default APEX version to 2.1.3 with HPX
- PR #3823²⁴²⁹ - Fix pkgconfig libs handling
- PR #3822²⁴³⁰ - Change includes in `hpx_wrap.cpp` to more specific includes
- PR #3821²⁴³¹ - Disable barrier_3792 test when networking is disabled
- PR #3820²⁴³² - Assorted CMake fixes
- PR #3815²⁴³³ - Removing left-over debug output
- PR #3814²⁴³⁴ - Allow setting default scheduler mode via the configuration database
- PR #3813²⁴³⁵ - Make the deprecation warnings issued by the old pp headers optional
- PR #3812²⁴³⁶ - Windows requires to handle symlinks to directories differently from those linking files
- PR #3811²⁴³⁷ - Clean up PP module and library skeleton
- PR #3806²⁴³⁸ - Moving include path configuration to before APEX
- PR #3804²⁴³⁹ - Fix latch
- PR #3803²⁴⁴⁰ - Update `hpxcxx` to look at lib64 and use python3
- PR #3802²⁴⁴¹ - Numa binding allocator
- PR #3801²⁴⁴² - Remove duplicated includes
- PR #3800²⁴⁴³ - Attempt to fix Posix context switching after lazy init changes
- PR #3798²⁴⁴⁴ - count and count_if accepts different iterator types

²⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/3833>

²⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/3831>

²⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3830>

²⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3828>

²⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3827>

²⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3826>

²⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3825>

²⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3823>

²⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3822>

²⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3821>

²⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/3820>

²⁴³³ <https://github.com/STELLAR-GROUP/hpx/pull/3815>

²⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3814>

²⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3813>

²⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3812>

²⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3811>

²⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3806>

²⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3804>

²⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3803>

²⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3802>

²⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3801>

²⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3800>

²⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3798>

- PR #3797²⁴⁴⁵ - Adding a couple of `override` keywords to overloaded virtual functions
- PR #3796²⁴⁴⁶ - Re-enable testing all schedulers in `shutdown_suspended_test`
- PR #3795²⁴⁴⁷ - Change `std::terminate` to `std::abort` in SIGSEGV handler
- PR #3794²⁴⁴⁸ - Fixing #3792
- PR #3793²⁴⁴⁹ - Extending `migrate_polymorphic_component` unit test
- PR #3791²⁴⁵⁰ - Change `throw()` to `noexcept`
- PR #3790²⁴⁵¹ - Remove deprecated options for 1.3.0 release
- PR #3789²⁴⁵² - Remove Boost filesystem compatibility header
- PR #3788²⁴⁵³ - Disabled even more spots that should not execute if networking is disabled
- PR #3787²⁴⁵⁴ - Bump minimal boost supported version to 1.61.0
- PR #3786²⁴⁵⁵ - Bump minimum required versions for 1.3.0 release
- PR #3785²⁴⁵⁶ - Explicitly set number of jobs for all ninja invocations on CircleCI
- PR #3784²⁴⁵⁷ - Fix leak and address sanitizer problems
- PR #3783²⁴⁵⁸ - Disabled even more spots that should not execute if networking is disabled
- PR #3782²⁴⁵⁹ - Cherry-picked tuple and `thread_init_data` fixes from #3701
- PR #3781²⁴⁶⁰ - Fix generic context coroutines after lazy stack allocation changes
- PR #3780²⁴⁶¹ - Rename hello world examples
- PR #3776²⁴⁶² - Sort algorithms now use the supplied chunker to determine the required minimal chunk size
- PR #3775²⁴⁶³ - Disable Boost auto-linking
- PR #3774²⁴⁶⁴ - Tag and push stable builds
- PR #3773²⁴⁶⁵ - Enable migration of polymorphic components
- PR #3771²⁴⁶⁶ - Fix link to stackoverflow in documentation
- PR #3770²⁴⁶⁷ - Replacing `constexpr` if in brace-serialization code

²⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3797>

²⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3796>

²⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3795>

²⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3794>

²⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3793>

²⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3791>

²⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3790>

²⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3789>

²⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3788>

²⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3787>

2455 <https://github.com/STELLAR-GROUP/hpx/pull/3786>2456 <https://github.com/STELLAR-GROUP/hpx/pull/3785>2457 <https://github.com/STELLAR-GROUP/hpx/pull/3784>2458 <https://github.com/STELLAR-GROUP/hpx/pull/3783>2459 <https://github.com/STELLAR-GROUP/hpx/pull/3782>2460 <https://github.com/STELLAR-GROUP/hpx/pull/3781>2461 <https://github.com/STELLAR-GROUP/hpx/pull/3780>2462 <https://github.com/STELLAR-GROUP/hpx/pull/3776>2463 <https://github.com/STELLAR-GROUP/hpx/pull/3775>2464 <https://github.com/STELLAR-GROUP/hpx/pull/3774>2465 <https://github.com/STELLAR-GROUP/hpx/pull/3773>2466 <https://github.com/STELLAR-GROUP/hpx/pull/3771>2467 <https://github.com/STELLAR-GROUP/hpx/pull/3770>

- PR #3769²⁴⁶⁸ - Fix SIGSEGV handler
- PR #3768²⁴⁶⁹ - Adding flags to scheduler allowing to control thread stealing and idle back-off
- PR #3767²⁴⁷⁰ - Fix help formatting in hpxrun.py
- PR #3765²⁴⁷¹ - Fix a couple of bugs in the thread test
- PR #3764²⁴⁷² - Workaround for SFNAE regression in msvc14.2
- PR #3762²⁴⁷³ - Prevent MSVC from prematurely instantiating things
- PR #3761²⁴⁷⁴ - Update python scripts to work with python 3
- PR #3760²⁴⁷⁵ - Fix callable vtable for GCC4.9
- PR #3759²⁴⁷⁶ - Rename PAGE_SIZE to PAGE_SIZE_ because AppleClang
- PR #3755²⁴⁷⁷ - Making sure locks are not held during suspension
- PR #3754²⁴⁷⁸ - Disable more code if networking is not available/not enabled
- PR #3752²⁴⁷⁹ - Move util::format implementation to source file
- PR #3751²⁴⁸⁰ - Fixing problems with lcos::barrier and iostreams
- PR #3750²⁴⁸¹ - Change error message to take into account use_guard_page setting
- PR #3749²⁴⁸² - Fix lifetime problem in run_as_hpx_thread
- PR #3748²⁴⁸³ - Fixed unusable behavior of the clang code analyzer.
- PR #3747²⁴⁸⁴ - Added PMIX_RANK to the defaults of HPX_WITH_PARCELPORT_MPI_ENV.
- PR #3745²⁴⁸⁵ - Introduced cache_aligned_data and cache_line_data helper structure
- PR #3742²⁴⁸⁶ - Remove more unused functionality from util/logging
- PR #3740²⁴⁸⁷ - Fix includes in partitioned vector tests
- PR #3739²⁴⁸⁸ - More fixes to make sure that std::flush really flushes all output
- PR #3737²⁴⁸⁹ - Fix potential shutdown problems
- PR #3736²⁴⁹⁰ - Fix guided_pool_executor after dataflow changes caused compilation fail

²⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3769>

²⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3768>

²⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3767>

²⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3765>

²⁴⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3764>

²⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3762>

²⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3761>

²⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3760>

²⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3759>

²⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3755>

2478 <https://github.com/STELLAR-GROUP/hpx/pull/3754>2479 <https://github.com/STELLAR-GROUP/hpx/pull/3752>2480 <https://github.com/STELLAR-GROUP/hpx/pull/3751>2481 <https://github.com/STELLAR-GROUP/hpx/pull/3750>2482 <https://github.com/STELLAR-GROUP/hpx/pull/3749>2483 <https://github.com/STELLAR-GROUP/hpx/pull/3748>2484 <https://github.com/STELLAR-GROUP/hpx/pull/3747>2485 <https://github.com/STELLAR-GROUP/hpx/pull/3745>2486 <https://github.com/STELLAR-GROUP/hpx/pull/3742>2487 <https://github.com/STELLAR-GROUP/hpx/pull/3740>2488 <https://github.com/STELLAR-GROUP/hpx/pull/3739>2489 <https://github.com/STELLAR-GROUP/hpx/pull/3737>2490 <https://github.com/STELLAR-GROUP/hpx/pull/3736>

- PR #3734²⁴⁹¹ - Limiting executor
- PR #3732²⁴⁹² - More constrained bound constructors
- PR #3730²⁴⁹³ - Attempt to fix deadlocks during component loading
- PR #3729²⁴⁹⁴ - Add latch member function count_up and reset, requested by hpxMP
- PR #3728²⁴⁹⁵ - Send even empty buffers on `hpx::endl` and `hpx::flush`
- PR #3727²⁴⁹⁶ - Adding example demonstrating how to customize the memory management for a component
- PR #3726²⁴⁹⁷ - Adding support for passing command line options through the `HPX_COMMANDLINE_OPTIONS` environment variable
- PR #3722²⁴⁹⁸ - Document known broken OpenMPI builds
- PR #3716²⁴⁹⁹ - Add barrier reset function, requested by hpxMP for reusing barrier
- PR #3715²⁵⁰⁰ - More work on functions and vtables
- PR #3714²⁵⁰¹ - Generate single-page HTML, PDF, manpage from documentation
- PR #3713²⁵⁰² - Updating default APEX version to 2.1.2
- PR #3712²⁵⁰³ - Update release procedure
- PR #3710²⁵⁰⁴ - Fix the C++11 build, after #3704
- PR #3709²⁵⁰⁵ - Move some component_registry functionality to source file
- PR #3708²⁵⁰⁶ - Ignore all locks while handling assertions
- PR #3707²⁵⁰⁷ - Remove obsolete hpx runtime executable
- PR #3705²⁵⁰⁸ - Fix and simplify `make_ready_future` overload sets
- PR #3704²⁵⁰⁹ - Reduce use of binders
- PR #3703²⁵¹⁰ - Ini
- PR #3702²⁵¹¹ - Fixing CUDA compiler errors
- PR #3700²⁵¹² - Added `barrier::increment` function to increase total number of thread
- PR #3697²⁵¹³ - One more attempt to fix migration...

²⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3734>

²⁴⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3732>

²⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3730>

²⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3729>

²⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3728>

²⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3727>

²⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3726>

²⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3722>

²⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3716>

²⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3715>

²⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3714>

²⁵⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3713>

²⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3712>

²⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3710>

²⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3709>

²⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3708>

²⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3707>

²⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3705>

²⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3704>

²⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3703>

²⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3702>

²⁵¹² <https://github.com/STELLAR-GROUP/hpx/pull/3700>

²⁵¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3697>

- PR #3694²⁵¹⁴ - Fixing component migration
- PR #3693²⁵¹⁵ - Print thread state when getting disallowed value in set_thread_state
- PR #3692²⁵¹⁶ - Only disable constexpr with clang-cuda, not nvcc+gcc
- PR #3691²⁵¹⁷ - Link with libsupc++ if needed for thread_local
- PR #3690²⁵¹⁸ - Remove thousands separators in set_operations_3442 to comply with C++11
- PR #3688²⁵¹⁹ - Decouple serialization from function vtables
- PR #3687²⁵²⁰ - Fix a couple of test failures
- PR #3686²⁵²¹ - Make sure tests.unit.build are run after install on CircleCI
- PR #3685²⁵²² - Revise quickstart CMakeLists.txt explanation
- PR #3684²⁵²³ - Provide concept emulation for Ranges-TS concepts
- PR #3683²⁵²⁴ - Ignore uninitialized chunks
- PR #3682²⁵²⁵ - Ignore uninitialized chunks. Check proper indices.
- PR #3680²⁵²⁶ - Ignore uninitialized chunks. Check proper range indices
- PR #3679²⁵²⁷ - Simplify basic action implementations
- PR #3678²⁵²⁸ - Making sure HPX_HAVE_LIBATOMIC is unset before checking
- PR #3677²⁵²⁹ - Fix generated full version number to be usable in expressions
- PR #3674²⁵³⁰ - Reduce functional utilities call depth
- PR #3672²⁵³¹ - Change new build system to use existing macros related to pseudo dependencies
- PR #3669²⁵³² - Remove indirection in function_ref when thread description is disabled
- PR #3668²⁵³³ - Unbreaking async_*cb* tests
- PR #3667²⁵³⁴ - Generate version.hpp
- PR #3665²⁵³⁵ - Enabling MPI parcelport for gitlab runners
- PR #3664²⁵³⁶ - making clang-tidy work properly again

²⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3694>

²⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3693>

²⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3692>

²⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3691>

²⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3690>

²⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3688>

²⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3687>

²⁵²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3686>

²⁵²² <https://github.com/STELLAR-GROUP/hpx/pull/3685>

²⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/3684>

²⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3683>

²⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3682>

²⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3680>

²⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3679>

²⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3678>

²⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3677>

²⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3674>

²⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3672>

²⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/3669>

²⁵³³ <https://github.com/STELLAR-GROUP/hpx/pull/3668>

²⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3667>

²⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3665>

²⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3664>

- PR #3662²⁵³⁷ - Attempt to fix exception handling
- PR #3661²⁵³⁸ - Move `lcos::latch` to source file
- PR #3660²⁵³⁹ - Fix accidentally explicit `gid_type` default constructor
- PR #3659²⁵⁴⁰ - Parallel executor latch
- PR #3658²⁵⁴¹ - Fixing `execution_parameters`
- PR #3657²⁵⁴² - Avoid dangling references in `wait_all`
- PR #3656²⁵⁴³ - Avoiding lifetime problems with `sync_put_parcel`
- PR #3655²⁵⁴⁴ - Fixing `nullptr` dereference inside of function
- PR #3652²⁵⁴⁵ - Attempt to fix `thread_map_type` definition with C++11
- PR #3650²⁵⁴⁶ - Allowing for end iterator being different from begin iterator
- PR #3649²⁵⁴⁷ - Added architecture identification to `cmake` to be able to detect timestamp support
- PR #3645²⁵⁴⁸ - Enabling sanitizers on gitlab runner
- PR #3644²⁵⁴⁹ - Attempt to tackle timeouts during startup
- PR #3642²⁵⁵⁰ - Cleanup parallel partitioners
- PR #3640²⁵⁵¹ - Dataflow now works with functions that return a reference
- PR #3637²⁵⁵² - Merging the executor-enabled overloads of `shared_future<>::then`
- PR #3633²⁵⁵³ - Replace deprecated boost endian macros
- PR #3632²⁵⁵⁴ - Add instructions on getting HPX to documentation
- PR #3631²⁵⁵⁵ - Simplify parcel creation
- PR #3630²⁵⁵⁶ - Small additions and fixes to release procedure
- PR #3629²⁵⁵⁷ - Modular pp
- PR #3627²⁵⁵⁸ - Implement `util::function_ref`
- PR #3626²⁵⁵⁹ - Fix `cancelable_action_client` example

²⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3662>

²⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3661>

²⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3660>

²⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3659>

²⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3658>

²⁵⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3657>

²⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3656>

²⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3655>

²⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3652>

²⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3650>

²⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3649>

²⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3645>

²⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3644>

²⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3642>

²⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3640>

²⁵⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3637>

²⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3633>

²⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3632>

²⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3631>

²⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3630>

²⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3629>

²⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3627>

²⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3626>

- PR #3625²⁵⁶⁰ - Added automatic serialization for simple structs (see #3034)
- PR #3624²⁵⁶¹ - Updating the default order of priority for `thread_description`
- PR #3621²⁵⁶² - Update copyright year and other small formatting fixes
- PR #3620²⁵⁶³ - Adding support for gitlab runner
- PR #3619²⁵⁶⁴ - Store debug logs and core dumps on CircleCI
- PR #3618²⁵⁶⁵ - Various optimizations
- PR #3617²⁵⁶⁶ - Fix link to the gpg key (#2)
- PR #3615²⁵⁶⁷ - Fix unused variable warnings with networking off
- PR #3614²⁵⁶⁸ - Restructuring counter data in scheduler to reduce false sharing
- PR #3613²⁵⁶⁹ - Adding support for gitlab runners
- PR #3610²⁵⁷⁰ - Don't wait for `stop_condition` in main thread
- PR #3608²⁵⁷¹ - Add inline keyword to `invalid_thread_id` definition for nvcc
- PR #3607²⁵⁷² - Adding configuration key that allows one to explicitly add a directory to the component search path
- PR #3606²⁵⁷³ - Add nvcc to exclude constexpress since is it not supported by nvcc
- PR #3605²⁵⁷⁴ - Add inline to definition of checkpoint stream operators to fix link error
- PR #3604²⁵⁷⁵ - Use format for string formatting
- PR #3603²⁵⁷⁶ - Improve the error message for using to less `MAX_CPU_COUNT`
- PR #3602²⁵⁷⁷ - Improve the error message for to small values of `MAX_CPU_COUNT`
- PR #3600²⁵⁷⁸ - Parallel executor aggregated
- PR #3599²⁵⁷⁹ - Making sure networking is disabled for default one-locality-runs
- PR #3596²⁵⁸⁰ - Store thread exit functions in `forward_list` instead of `deque` to avoid allocations
- PR #3590²⁵⁸¹ - Fix typo/mistake in thread queue `cleanup_terminated`
- PR #3588²⁵⁸² - Fix formatting errors in launching_and_configuring_hpx_applications.rst

2560 <https://github.com/STELLAR-GROUP/hpx/pull/3625>

2561 <https://github.com/STELLAR-GROUP/hpx/pull/3624>

2562 <https://github.com/STELLAR-GROUP/hpx/pull/3621>

2563 <https://github.com/STELLAR-GROUP/hpx/pull/3620>

2564 <https://github.com/STELLAR-GROUP/hpx/pull/3619>

2565 <https://github.com/STELLAR-GROUP/hpx/pull/3618>

2566 <https://github.com/STELLAR-GROUP/hpx/pull/3617>

2567 <https://github.com/STELLAR-GROUP/hpx/pull/3615>

2568 <https://github.com/STELLAR-GROUP/hpx/pull/3614>

2569 <https://github.com/STELLAR-GROUP/hpx/pull/3613>

2570 <https://github.com/STELLAR-GROUP/hpx/pull/3610>

2571 <https://github.com/STELLAR-GROUP/hpx/pull/3608>

2572 <https://github.com/STELLAR-GROUP/hpx/pull/3607>

2573 <https://github.com/STELLAR-GROUP/hpx/pull/3606>

2574 <https://github.com/STELLAR-GROUP/hpx/pull/3605>

2575 <https://github.com/STELLAR-GROUP/hpx/pull/3604>

2576 <https://github.com/STELLAR-GROUP/hpx/pull/3603>

2577 <https://github.com/STELLAR-GROUP/hpx/pull/3602>

2578 <https://github.com/STELLAR-GROUP/hpx/pull/3600>

2579 <https://github.com/STELLAR-GROUP/hpx/pull/3599>

2580 <https://github.com/STELLAR-GROUP/hpx/pull/3596>

2581 <https://github.com/STELLAR-GROUP/hpx/pull/3590>

2582 <https://github.com/STELLAR-GROUP/hpx/pull/3588>

- PR #3586²⁵⁸³ - Make bind propagate value category
- PR #3585²⁵⁸⁴ - Extend Cmake for building hpx as distribution packages (refs #3575)
- PR #3584²⁵⁸⁵ - Untangle function storage from object pointer
- PR #3582²⁵⁸⁶ - Towards Modularized HPX
- PR #3580²⁵⁸⁷ - Remove extra || in merge.hpp
- PR #3577²⁵⁸⁸ - Partially revert “Remove vtable empty flag”
- PR #3576²⁵⁸⁹ - Make sure empty startup/shutdown functions are not being used
- PR #3574²⁵⁹⁰ - Make sure DATAPAR settings are conveyed to depending projects
- PR #3573²⁵⁹¹ - Make sure HPX is usable with latest released version of Vc (V1.4.1)
- PR #3572²⁵⁹² - Adding test ensuring ticket 3565 is fixed
- PR #3571²⁵⁹³ - Make empty [unique_]function vtable non-dependent
- PR #3566²⁵⁹⁴ - Fix compilation with dynamic bitset for CPU masks
- PR #3563²⁵⁹⁵ - Drop util::[unique_]function target_type
- PR #3562²⁵⁹⁶ - Removing the target suffixes
- PR #3561²⁵⁹⁷ - Replace executor traits return type deduction (keep non-SFINAE)
- PR #3557²⁵⁹⁸ - Replace the last usages of boost::atomic
- PR #3556²⁵⁹⁹ - Replace boost::scoped_array with std::unique_ptr
- PR #3552²⁶⁰⁰ - (Re)move APEX readme
- PR #3548²⁶⁰¹ - Replace boost::scoped_ptr with std::unique_ptr
- PR #3547²⁶⁰² - Remove last use of Boost.Signals2
- PR #3544²⁶⁰³ - Post 1.2.0 version bumps
- PR #3543²⁶⁰⁴ - added Ubuntu dependency list to readme
- PR #3531²⁶⁰⁵ - Warnings, warnings...

²⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3586>

²⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3585>

²⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3584>

²⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3582>

²⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3580>

²⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3577>

²⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3576>

²⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3574>

²⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3573>

²⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3572>

²⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3571>

²⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3566>

²⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3563>

²⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3562>

²⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3561>

²⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3557>

²⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3556>

²⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3552>

²⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3548>

²⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3547>

²⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3544>

²⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3543>

²⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3531>

- PR #3527²⁶⁰⁶ - Add CircleCI filter for building all tags
- PR #3525²⁶⁰⁷ - Segmented algorithms
- PR #3517²⁶⁰⁸ - Replace boost::regex with C++11 <regex>
- PR #3514²⁶⁰⁹ - Cleaning up the build system
- PR #3505²⁶¹⁰ - Fixing type attribute warning for transfer_action
- PR #3504²⁶¹¹ - Add support for rpm packaging
- PR #3499²⁶¹² - Improving spinlock pools
- PR #3498²⁶¹³ - Remove thread specific ptr
- PR #3486²⁶¹⁴ - Fix comparison for expect_connecting_localities config entry
- PR #3469²⁶¹⁵ - Enable (existing) code for extracting stack pointer on Power platform

2.10.12 HPX V1.2.1 (Feb 19, 2019)

General changes

This is a bugfix release. It contains the following changes:

- Fix compilation on ARM, s390x and 32-bit architectures.
- Fix a critical bug in the future implementation.
- Fix several problems in the CMake configuration which affects external projects.
- Add support for Boost 1.69.0.

Closed issues

- Issue #3638²⁶¹⁶ - Build HPX 1.2 with boost 1.69
- Issue #3635²⁶¹⁷ - Non-deterministic crashing on Stampede2
- Issue #3550²⁶¹⁸ - 1>e:000workhpxsrcthrow_exception.cpp(54): error C2440: ‘<function-style-cast>’: cannot convert from ‘boost::system::error_code’ to ‘hpx::exception’
- Issue #3549²⁶¹⁹ - HPX 1.2.0 does not build on i686, but release candidate did
- Issue #3511²⁶²⁰ - Build on s390x fails
- Issue #3509²⁶²¹ - Build on armv7l fails

²⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3527>

²⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3525>

²⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3517>

²⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3514>

²⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3505>

²⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3504>

²⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/3499>

²⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3498>

²⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3486>

²⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3469>

²⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3638>

²⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3635>

²⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3550>

²⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3549>

²⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3511>

²⁶²¹ <https://github.com/STELLAR-GROUP/hpx/issues/3509>

Closed pull requests

- PR #3695²⁶²² - Don't install CMake templates and packaging files
- PR #3666²⁶²³ - Fixing yet another race in future_data
- PR #3663²⁶²⁴ - Fixing race between setting and getting the value inside future_data
- PR #3648²⁶²⁵ - Adding timestamp option for S390x platform
- PR #3647²⁶²⁶ - Blind attempt to fix warnings issued by gcc V9
- PR #3611²⁶²⁷ - Include GNUInstallDirs earlier to have it available for subdirectories
- PR #3595²⁶²⁸ - Use GNUInstallDirs lib path in pkgconfig config file
- PR #3593²⁶²⁹ - Add include(GNUInstallDirs) to HPXMacros.cmake
- PR #3591²⁶³⁰ - Fix compilation error on arm7 architecture. Compiles and runs on Fedora 29 on Pi 3.
- PR #3558²⁶³¹ - Adding constructor *exception(boost::system::error_code const&)*
- PR #3555²⁶³² - cmake: make install locations configurable
- PR #3551²⁶³³ - Fix uint64_t causing compilation fail on i686

2.10.13 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 CMake²⁶³⁴ or `pkg-config`. The old behaviour can be restored by setting `HPX_WITH_DYNAMIC_HPX_MAIN=OFF` during CMake²⁶³⁵ 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.

²⁶²² <https://github.com/STELLAR-GROUP/hpx/pull/3695>

²⁶²³ <https://github.com/STELLAR-GROUP/hpx/pull/3666>

²⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3663>

²⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3648>

²⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3647>

²⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3611>

²⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3595>

²⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3593>

²⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3591>

²⁶³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3558>

²⁶³² <https://github.com/STELLAR-GROUP/hpx/pull/3555>

²⁶³³ <https://github.com/STELLAR-GROUP/hpx/pull/3551>

²⁶³⁴ <https://www.cmake.org>

²⁶³⁵ <https://www.cmake.org>

- 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 `a11` 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 [CMake²⁶³⁶](#) 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 [CMake²⁶³⁷](#) 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 [CMake²⁶³⁸](#) configuration.
- It is now possible to register callback functions for *HPX* worker thread events. Callbacks can be registered for starting and stopping worker threads, and for when errors occur.

Breaking changes

- The implementation of `hpx_main.hpp` has changed. If you are using custom Makefiles you will need to make changes. Please see the documentation on [using Makefiles](#) for more details.
- The default value of `--hpx:threads` has changed from `a11` 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 #3266²⁶³⁹](#).
- 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`.

²⁶³⁶ <https://www.cmake.org>

²⁶³⁷ <https://www.cmake.org>

²⁶³⁸ <https://www.cmake.org>

²⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3266>

Closed issues

- Issue #3538²⁶⁴⁰ - numa handling incorrect for hwloc 2
- Issue #3533²⁶⁴¹ - Cmake version 3.5.1does not work (git ff26b35 2018-11-06)
- Issue #3526²⁶⁴² - Failed building hpx-1.2.0-rc1 on Ubuntu16.04 x86-64 Virtualbox VM
- Issue #3512²⁶⁴³ - Build on aarch64 fails
- Issue #3475²⁶⁴⁴ - HPX fails to link if the MPI parcelport is enabled
- Issue #3462²⁶⁴⁵ - CMake configuration shows a minor and inconsequential failure to create a symlink
- Issue #3461²⁶⁴⁶ - Compilation Problems with the most recent Clang
- Issue #3460²⁶⁴⁷ - Deadlock when create_partitioner fails (assertion fails) in debug mode
- Issue #3455²⁶⁴⁸ - HPX build failing with HWLOC errors on POWER8 with hwloc 1.8
- Issue #3438²⁶⁴⁹ - HPX no longer builds on IBM POWER8
- Issue #3426²⁶⁵⁰ - hpx build failed on MacOS
- Issue #3424²⁶⁵¹ - CircleCI builds broken for forked repositories
- Issue #3422²⁶⁵² - Benchmarks in tests.performance.local are not run nightly
- Issue #3408²⁶⁵³ - CMake Targets for HPX
- Issue #3399²⁶⁵⁴ - processing unit out of bounds
- Issue #3395²⁶⁵⁵ - Floating point bug in hpx/runtime/threads/policies/scheduler_base.hpp
- Issue #3378²⁶⁵⁶ - compile error with lcos::communicator
- Issue #3376²⁶⁵⁷ - Failed to build HPX with APEX using clang
- Issue #3366²⁶⁵⁸ - Adapted Safe_Object example fails for -hpx:threads > 1
- Issue #3360²⁶⁵⁹ - Segmentation fault when passing component id as parameter
- Issue #3358²⁶⁶⁰ - HPX runtime hangs after multiple (~thousands) start-stop sequences
- Issue #3352²⁶⁶¹ - Support TCP provider in libfabric ParcelPort

²⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3538>

²⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/3533>

²⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/3526>

²⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/3512>

²⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3475>

²⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3462>

²⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3461>

²⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3460>

²⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3455>

²⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3438>

²⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3426>

²⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/3424>

²⁶⁵² <https://github.com/STELLAR-GROUP/hpx/issues/3422>

²⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/3408>

²⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3399>

²⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3395>

²⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3378>

²⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3376>

²⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3366>

²⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3360>

²⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3358>

²⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/3352>

- Issue #3342²⁶⁶² - undefined reference to __atomic_load_16
- Issue #3339²⁶⁶³ - setting command line options/flags from init cfg is not obvious
- Issue #3325²⁶⁶⁴ - AGAS migrates components prematurely
- Issue #3321²⁶⁶⁵ - hpx bad_parameter handling is awful
- Issue #3318²⁶⁶⁶ - Benchmarks fail to build with C++11
- Issue #3304²⁶⁶⁷ - hpx::threads::run_as_hpx_thread does not properly handle exceptions
- Issue #3300²⁶⁶⁸ - Setting pu step or offset results in no threads in default pool
- Issue #3297²⁶⁶⁹ - Crash with APEX when running Phylanx lra_csv with > 1 thread
- Issue #3296²⁶⁷⁰ - Building HPX with APEX configuration gives compiler warnings
- Issue #3290²⁶⁷¹ - make tests failing at hello_world_component
- Issue #3285²⁶⁷² - possible compilation error when “using namespace std;” is defined before including “hpx” headers files
- Issue #3280²⁶⁷³ - HPX fails on OSX
- Issue #3272²⁶⁷⁴ - CircleCI does not upload generated docker image any more
- Issue #3270²⁶⁷⁵ - Error when compiling CUDA examples
- Issue #3267²⁶⁷⁶ - tests.unit.host_.block_allocator fails occasionally
- Issue #3264²⁶⁷⁷ - Possible move to Sphinx for documentation
- Issue #3263²⁶⁷⁸ - Documentation improvements
- Issue #3259²⁶⁷⁹ - set_parcel_write_handler test fails occasionally
- Issue #3258²⁶⁸⁰ - Links to source code in documentation are broken
- Issue #3247²⁶⁸¹ - Rare tests.unit.host_.block_allocator test failure on 1.1.0-rc1
- Issue #3244²⁶⁸² - Slowing down and speeding up an interval_timer
- Issue #3215²⁶⁸³ - Cannot build both tests and examples on MSVC with pseudo-dependencies enabled
- Issue #3195²⁶⁸⁴ - Unnecessary customization point route causing performance penalty

²⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/3342>

²⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/3339>

²⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3325>

²⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3321>

²⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3318>

²⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3304>

²⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3300>

²⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3297>

²⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3296>

²⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/3290>

²⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/3285>

²⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/3280>

²⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3272>

²⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3270>

²⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3267>

²⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3264>

²⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3263>

²⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3259>

²⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3258>

²⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/3247>

²⁶⁸² <https://github.com/STELLAR-GROUP/hpx/issues/3244>

²⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/3215>

²⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3195>

- Issue #3088²⁶⁸⁵ - A strange thing in parallel::sort.
- Issue #2650²⁶⁸⁶ - libfabric support for passive endpoints
- Issue #1205²⁶⁸⁷ - TSS is broken

Closed pull requests

- PR #3542²⁶⁸⁸ - Fix numa lookup from pu when using hwloc 2.x
- PR #3541²⁶⁸⁹ - Fixing the build system of the MPI parcelport
- PR #3540²⁶⁹⁰ - Updating HPX people section
- PR #3539²⁶⁹¹ - Splitting test to avoid OOM on CircleCI
- PR #3537²⁶⁹² - Fix guided exec
- PR #3536²⁶⁹³ - Updating grants which support the LSU team
- PR #3535²⁶⁹⁴ - Fix hiding of docker credentials
- PR #3534²⁶⁹⁵ - Fixing #3533
- PR #3532²⁶⁹⁶ - fixing minor doc typo --hpx:print-counter-at arg
- PR #3530²⁶⁹⁷ - Changing APEX default tag to v2.1.0
- PR #3529²⁶⁹⁸ - Remove leftover security options and documentation
- PR #3528²⁶⁹⁹ - Fix hwloc version check
- PR #3524²⁷⁰⁰ - Do not build guided pool examples with older GCC compilers
- PR #3523²⁷⁰¹ - Fix logging regression
- PR #3522²⁷⁰² - Fix more warnings
- PR #3521²⁷⁰³ - Fixing argument handling in induction and reduction clauses for parallel::for_loop
- PR #3520²⁷⁰⁴ - Remove docs symlink and versioned docs folders
- PR #3519²⁷⁰⁵ - hpxMP release
- PR #3518²⁷⁰⁶ - Change all steps to use new docker image on CircleCI

²⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3088>

²⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2650>

²⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1205>

²⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3542>

²⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3541>

²⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3540>

²⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3539>

²⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3537>

²⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3536>

²⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3535>

²⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3534>

²⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3532>

²⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3530>

²⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3529>

²⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3528>

²⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3524>

²⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3523>

²⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3522>

²⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3521>

²⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3520>

²⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3519>

²⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3518>

- PR #3516²⁷⁰⁷ - Drop usage of deprecated facilities removed in C++17
- PR #3515²⁷⁰⁸ - Remove remaining uses of Boost.TypeTraits
- PR #3513²⁷⁰⁹ - Fixing a CMake problem when trying to use libfabric
- PR #3508²⁷¹⁰ - Remove memory_block component
- PR #3507²⁷¹¹ - Propagating the MPI compile definitions to all relevant targets
- PR #3503²⁷¹² - Update documentation colors and logo
- PR #3502²⁷¹³ - Fix bogus `throws` bindings in scheduled_thread_pool_impl
- PR #3501²⁷¹⁴ - Split parallel::remove_if tests to avoid OOM on CircleCI
- PR #3500²⁷¹⁵ - Support NONAMEPREFIX in add_hpx_library()
- PR #3497²⁷¹⁶ - Note that cuda support requires cmake 3.9
- PR #3495²⁷¹⁷ - Fixing dataflow
- PR #3493²⁷¹⁸ - Remove deprecated options for 1.2.0 part 2
- PR #3492²⁷¹⁹ - Add CUDA_LINK_LIBRARIES_KEYWORD to allow PRIVATE keyword in linkage t...
- PR #3491²⁷²⁰ - Changing Base docker image
- PR #3490²⁷²¹ - Don't create tasks immediately with hpx::apply
- PR #3489²⁷²² - Remove deprecated options for 1.2.0
- PR #3488²⁷²³ - Revert "Use BUILD_INTERFACE generator expression to fix cmake flag exports"
- PR #3487²⁷²⁴ - Revert "Fixing type attribute warning for transfer_action"
- PR #3485²⁷²⁵ - Use BUILD_INTERFACE generator expression to fix cmake flag exports
- PR #3483²⁷²⁶ - Fixing type attribute warning for transfer_action
- PR #3481²⁷²⁷ - Remove unused variables
- PR #3480²⁷²⁸ - Towards a more lightweight transfer action
- PR #3479²⁷²⁹ - Fix FLAGS - Use correct version of target_compile_options

²⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3516>

²⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3515>

²⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3513>

²⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3508>

²⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3507>

²⁷¹² <https://github.com/STELLAR-GROUP/hpx/pull/3503>

²⁷¹³ <https://github.com/STELLAR-GROUP/hpx/pull/3502>

²⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3501>

²⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3500>

²⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3497>

²⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3495>

²⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3493>

²⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3492>

²⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3491>

²⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3490>

²⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/3489>

²⁷²³ <https://github.com/STELLAR-GROUP/hpx/pull/3488>

²⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3487>

²⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3485>

²⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3483>

²⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3481>

²⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3480>

²⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3479>

- PR #3478²⁷³⁰ - Making sure the application's exit code is properly propagated back to the OS
- PR #3476²⁷³¹ - Don't print docker credentials as part of the environment.
- PR #3473²⁷³² - Fixing invalid cmake code if no jemalloc prefix was given
- PR #3472²⁷³³ - Attempting to work around recent clang test compilation failures
- PR #3471²⁷³⁴ - Enable jemalloc on windows
- PR #3470²⁷³⁵ - Updates readme
- PR #3468²⁷³⁶ - Avoid hang if there is an exception thrown during startup
- PR #3467²⁷³⁷ - Add compiler specific fallthrough attributes if C++17 attribute is not available
- PR #3466²⁷³⁸ - - bugfix : fix compilation with llvm-7.0
- PR #3465²⁷³⁹ - This patch adds various optimizations extracted from the thread_local_allocator work
- PR #3464²⁷⁴⁰ - Check for forked repos in CircleCI docker push step
- PR #3463²⁷⁴¹ - - cmake : create the parent directory before symlinking
- PR #3459²⁷⁴² - Remove unused/incomplete functionality from util/logging
- PR #3458²⁷⁴³ - Fix a problem with scope of CMAKE_CXX_FLAGS and hpx_add_compile_flag
- PR #3457²⁷⁴⁴ - Fixing more size_t -> int16_t (and similar) warnings
- PR #3456²⁷⁴⁵ - Add #ifdefs to topology.cpp to support old hwloc versions again
- PR #3454²⁷⁴⁶ - Fixing warnings related to silent conversion of size_t -> int16_t
- PR #3451²⁷⁴⁷ - Add examples as unit tests
- PR #3450²⁷⁴⁸ - Constexpr-fying bind and other functional facilities
- PR #3446²⁷⁴⁹ - Fix some thread suspension timeouts
- PR #3445²⁷⁵⁰ - Fix various warnings
- PR #3443²⁷⁵¹ - Only enable service pool config options if pools are enabled
- PR #3441²⁷⁵² - Fix missing closing brackets in documentation

²⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3478>

²⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3476>

²⁷³² <https://github.com/STELLAR-GROUP/hpx/pull/3473>

²⁷³³ <https://github.com/STELLAR-GROUP/hpx/pull/3472>

²⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3471>

²⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3470>

²⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3468>

²⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3467>

²⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3466>

²⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3465>

2740 <https://github.com/STELLAR-GROUP/hpx/pull/3464>2741 <https://github.com/STELLAR-GROUP/hpx/pull/3463>2742 <https://github.com/STELLAR-GROUP/hpx/pull/3459>2743 <https://github.com/STELLAR-GROUP/hpx/pull/3458>2744 <https://github.com/STELLAR-GROUP/hpx/pull/3457>2745 <https://github.com/STELLAR-GROUP/hpx/pull/3456>2746 <https://github.com/STELLAR-GROUP/hpx/pull/3454>2747 <https://github.com/STELLAR-GROUP/hpx/pull/3451>2748 <https://github.com/STELLAR-GROUP/hpx/pull/3450>2749 <https://github.com/STELLAR-GROUP/hpx/pull/3446>2750 <https://github.com/STELLAR-GROUP/hpx/pull/3445>2751 <https://github.com/STELLAR-GROUP/hpx/pull/3443>2752 <https://github.com/STELLAR-GROUP/hpx/pull/3441>

- PR #3439²⁷⁵³ - Use correct MPI CXX libraries for MPI parcelport
- PR #3436²⁷⁵⁴ - Add projection function to find_* (and fix very bad bug)
- PR #3435²⁷⁵⁵ - Fixing 1205
- PR #3434²⁷⁵⁶ - Fix threads cores
- PR #3433²⁷⁵⁷ - Add Heise Online to release announcement list
- PR #3432²⁷⁵⁸ - Don't track task dependencies for distributed runs
- PR #3431²⁷⁵⁹ - Circle CI setting changes for hpxMP
- PR #3430²⁷⁶⁰ - Fix unused params warning
- PR #3429²⁷⁶¹ - One thread per core
- PR #3428²⁷⁶² - This suppresses a deprecation warning that is being issued by MSVC 19.15.26726
- PR #3427²⁷⁶³ - Fixes #3426
- PR #3425²⁷⁶⁴ - Use source cache and workspace between job steps on CircleCI
- PR #3421²⁷⁶⁵ - Add CDash timing output to future overhead test (for graphs)
- PR #3420²⁷⁶⁶ - Add guided_pool_executor
- PR #3419²⁷⁶⁷ - Fix typo in CircleCI config
- PR #3418²⁷⁶⁸ - Add sphinx documentation
- PR #3415²⁷⁶⁹ - Scheduler NUMA hint and shared priority scheduler
- PR #3414²⁷⁷⁰ - Adding step to synchronize the APEX release
- PR #3413²⁷⁷¹ - Fixing multiple defines of APEX_HAVE_HPX
- PR #3412²⁷⁷² - Fixes linking with libhpx_wrap error with BSD and Windows based systems
- PR #3410²⁷⁷³ - Fix typo in CMakeLists.txt
- PR #3409²⁷⁷⁴ - Fix brackets and indentation in existing_performance_counters.qbk
- PR #3407²⁷⁷⁵ - Fix unused param and extra ; warnings emitted by gcc 8.x

²⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3439>

²⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3436>

²⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3435>

²⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3434>

²⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3433>

²⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3432>

²⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3431>

²⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3430>

²⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3429>

²⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3428>

²⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3427>

²⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3425>

²⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3421>

²⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3420>

²⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3419>

²⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3418>

²⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3415>

²⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3414>

²⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3413>

²⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3412>

²⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3410>

²⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3409>

²⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3407>

- PR #3406²⁷⁷⁶ - Adding thread local allocator and use it for future shared states
- PR #3405²⁷⁷⁷ - Adding DHPX_HAVE_THREAD_LOCAL_STORAGE=ON to builds
- PR #3404²⁷⁷⁸ - fixing multiple definition of main() in linux
- PR #3402²⁷⁷⁹ - Allow debug option to be enabled only for Linux systems with dynamic main on
- PR #3401²⁷⁸⁰ - Fix cuda_future_helper.h when compiling with C++11
- PR #3400²⁷⁸¹ - Fix floating point exception scheduler_base idle backoff
- PR #3398²⁷⁸² - Atomic future state
- PR #3397²⁷⁸³ - Fixing code for older gcc versions
- PR #3396²⁷⁸⁴ - Allowing to register thread event functions (start/stop/error)
- PR #3394²⁷⁸⁵ - Fix small mistake in primary_namespace_server.cpp
- PR #3393²⁷⁸⁶ - Explicitly instantiate configured schedulers
- PR #3392²⁷⁸⁷ - Add performance counters background overhead and background work duration
- PR #3391²⁷⁸⁸ - Adapt integration of HPXMP to latest build system changes
- PR #3390²⁷⁸⁹ - Make AGAS measurements optional
- PR #3389²⁷⁹⁰ - Fix deadlock during shutdown
- PR #3388²⁷⁹¹ - Add several functionalities allowing to optimize synchronous action invocation
- PR #3387²⁷⁹² - Add cmake option to opt out of fail-compile tests
- PR #3386²⁷⁹³ - Adding support for boost::container::small_vector to dataflow
- PR #3385²⁷⁹⁴ - Adds Debug option for hpx initializing from main
- PR #3384²⁷⁹⁵ - This hopefully fixes two tests that occasionally fail
- PR #3383²⁷⁹⁶ - Making sure thread local storage is enable for hpxMP
- PR #3382²⁷⁹⁷ - Fix usage of HPX_CAPTURE together with default value capture [=]
- PR #3381²⁷⁹⁸ - Replace undefined instantiations of uniform_int_distribution

²⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3406>

²⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3405>

²⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3404>

²⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3402>

²⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3401>

²⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3400>

²⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3398>

²⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3397>

²⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3396>

²⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3394>

²⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3393>

²⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3392>

²⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3391>

²⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3390>

²⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3389>

²⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3388>

²⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3387>

²⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3386>

²⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3385>

²⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3384>

²⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3383>

²⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3382>

²⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3381>

- PR #3380²⁷⁹⁹ - Add missing semicolons to uses of HPX_COMPILER_FENCE
- PR #3379²⁸⁰⁰ - Fixing #3378
- PR #3377²⁸⁰¹ - Adding build system support to integrate hpxmp into hpx at the user's machine
- PR #3375²⁸⁰² - Replacing wrapper for __libc_start_main with main
- PR #3374²⁸⁰³ - Adds hpx_wrap to HPX_LINK_LIBRARIES which links only when specified.
- PR #3373²⁸⁰⁴ - Forcing cache settings in HPXConfig.cmake to guarantee updated values
- PR #3372²⁸⁰⁵ - Fix some more c++11 build problems
- PR #3371²⁸⁰⁶ - Adds HPX_LINKER_FLAGS to HPX applications without editing their source codes
- PR #3370²⁸⁰⁷ - util::format: add typeSpecifier specializations for %!s(MISSING) and %!!(MISSING)s
- PR #3369²⁸⁰⁸ - Adding configuration option to allow explicit disable of the new hpx_main feature on Linux
- PR #3368²⁸⁰⁹ - Updates doc with recent hpx_wrap implementation
- PR #3367²⁸¹⁰ - Adds Mac OS implementation to hpx_main.hpp
- PR #3365²⁸¹¹ - Fix order of hpx libs in HPX_CONF_LIBRARIES.
- PR #3363²⁸¹² - Apex fixing null wrapper
- PR #3361²⁸¹³ - Making sure all parcels get destroyed on an HPX thread (TCP pp)
- PR #3359²⁸¹⁴ - Feature/improveerrorforcompiler
- PR #3357²⁸¹⁵ - Static/dynamic executable implementation
- PR #3355²⁸¹⁶ - Reverting changes introduced by #3283 as those make applications hang
- PR #3354²⁸¹⁷ - Add external dependencies to HPX_LIBRARY_DIR
- PR #3353²⁸¹⁸ - Fix libfabric tcp
- PR #3351²⁸¹⁹ - Move obsolete header to tests directory.
- PR #3350²⁸²⁰ - Renaming two functions to avoid problem described in #3285
- PR #3349²⁸²¹ - Make idle backoff exponential with maximum sleep time

²⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3380>

²⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3379>

²⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3377>

²⁸⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3375>

²⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3374>

²⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3373>

²⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3372>

²⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3371>

²⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3370>

2808 <https://github.com/STELLAR-GROUP/hpx/pull/3369>2809 <https://github.com/STELLAR-GROUP/hpx/pull/3368>2810 <https://github.com/STELLAR-GROUP/hpx/pull/3367>2811 <https://github.com/STELLAR-GROUP/hpx/pull/3365>2812 <https://github.com/STELLAR-GROUP/hpx/pull/3363>2813 <https://github.com/STELLAR-GROUP/hpx/pull/3361>2814 <https://github.com/STELLAR-GROUP/hpx/pull/3359>2815 <https://github.com/STELLAR-GROUP/hpx/pull/3357>2816 <https://github.com/STELLAR-GROUP/hpx/pull/3355>2817 <https://github.com/STELLAR-GROUP/hpx/pull/3354>2818 <https://github.com/STELLAR-GROUP/hpx/pull/3353>2819 <https://github.com/STELLAR-GROUP/hpx/pull/3351>2820 <https://github.com/STELLAR-GROUP/hpx/pull/3350>2821 <https://github.com/STELLAR-GROUP/hpx/pull/3349>

- PR #3347²⁸²² - Replace *simple_component** with *component** in the Documentation
- PR #3346²⁸²³ - Fix CMakeLists.txt example in quick start
- PR #3345²⁸²⁴ - Fix automatic setting of HPX_MORE_THAN_64_THREADS
- PR #3344²⁸²⁵ - Reduce amount of information printed for unknown command line options
- PR #3343²⁸²⁶ - Safeguard HPX against destruction in global contexts
- PR #3341²⁸²⁷ - Allowing for all command line options to be used as configuration settings
- PR #3340²⁸²⁸ - Always convert inspect results to JUnit XML
- PR #3336²⁸²⁹ - Only run docker push on master on CircleCI
- PR #3335²⁸³⁰ - Update description of hpx.os_threads config parameter.
- PR #3334²⁸³¹ - Making sure early logging settings don't get mixed with others
- PR #3333²⁸³² - Update CMake links and versions in documentation
- PR #3332²⁸³³ - Add notes on target suffixes to CMake documentation
- PR #3331²⁸³⁴ - Add quickstart section to documentation
- PR #3330²⁸³⁵ - Rename resource_partitioner test to avoid conflicts with pseudodependencies
- PR #3328²⁸³⁶ - Making sure object is pinned while executing actions, even if action returns a future
- PR #3327²⁸³⁷ - Add missing std::forward to tuple.hpp
- PR #3326²⁸³⁸ - Make sure logging is up and running while modules are being discovered.
- PR #3324²⁸³⁹ - Replace C++14 overload of std::equal with C++11 code.
- PR #3323²⁸⁴⁰ - Fix a missing apex thread data (wrapper) initialization
- PR #3320²⁸⁴¹ - Adding support for -std=c++2a (define *HPX_WITH_CXX2A=On*)
- PR #3319²⁸⁴² - Replacing C++14 feature with equivalent C++11 code
- PR #3317²⁸⁴³ - Fix compilation with VS 15.7.1 and /std:c++latest
- PR #3316²⁸⁴⁴ - Fix includes for 1d_stencil_*_omp examples

²⁸²² <https://github.com/STELLAR-GROUP/hpx/pull/3347>²⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/3346>²⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3345>²⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3344>²⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3343>²⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3341>²⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3340>²⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3336>²⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3335>²⁸³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3334>²⁸³² <https://github.com/STELLAR-GROUP/hpx/pull/3333>²⁸³³ <https://github.com/STELLAR-GROUP/hpx/pull/3332>²⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3331>²⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3330>²⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3328>²⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3327>²⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3326>²⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3324>²⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3323>²⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3320>²⁸⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3319>²⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3317>²⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3316>

- PR #3314²⁸⁴⁵ - Remove some unused parameter warnings
- PR #3313²⁸⁴⁶ - Fix pu-step and pu-offset command line options
- PR #3312²⁸⁴⁷ - Add conversion of inspect reports to JUnit XML
- PR #3311²⁸⁴⁸ - Fix escaping of closing braces in format specification syntax
- PR #3310²⁸⁴⁹ - Don't overwrite user settings with defaults in registration database
- PR #3309²⁸⁵⁰ - Fixing potential stack overflow for dataflow
- PR #3308²⁸⁵¹ - This updates the .clang-format configuration file to utilize newer features
- PR #3306²⁸⁵² - Marking migratable objects in their gid to allow not handling migration in AGAS
- PR #3305²⁸⁵³ - Add proper exception handling to run_as_hpx_thread
- PR #3303²⁸⁵⁴ - Changed std::rand to a better inbuilt PRNG Generator
- PR #3302²⁸⁵⁵ - All non-migratable (simple) components now encode their lva and component type in their gid
- PR #3301²⁸⁵⁶ - Add nullptr_t overloads to resource partitioner
- PR #3298²⁸⁵⁷ - Apex task wrapper memory bug
- PR #3295²⁸⁵⁸ - Fix mistakes after merge of CircleCI config
- PR #3294²⁸⁵⁹ - Fix partitioned vector include in partitioned_vector_find tests
- PR #3293²⁸⁶⁰ - Adding emplace support to promise and make_ready_future
- PR #3292²⁸⁶¹ - Add new cuda kernel synchronization with hpx::future demo
- PR #3291²⁸⁶² - Fixes #3290
- PR #3289²⁸⁶³ - Fixing Docker image creation
- PR #3288²⁸⁶⁴ - Avoid allocating shared state for wait_all
- PR #3287²⁸⁶⁵ - Fixing /scheduler/utilization/instantaneous performance counter
- PR #3286²⁸⁶⁶ - dataflow() and future::then() use sync policy where possible
- PR #3284²⁸⁶⁷ - Background thread can use relaxed atomics to manipulate thread state

²⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3314>

²⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3313>

²⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3312>

²⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3311>

²⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3310>

²⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3309>

²⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3308>

²⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3306>

²⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3305>

²⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3303>

²⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3302>

²⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3301>

²⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3298>

²⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3295>

²⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3294>

²⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3293>

²⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3292>

²⁸⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3291>

²⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3289>

²⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3288>

²⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3287>

²⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3286>

²⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3284>

- PR #3283²⁸⁶⁸ - Do not unwrap ready future
- PR #3282²⁸⁶⁹ - Fix virtual method override warnings in static schedulers
- PR #3281²⁸⁷⁰ - Disable set_area_membind_nodeset for OSX
- PR #3279²⁸⁷¹ - Add two variations to the future_overhead benchmark
- PR #3278²⁸⁷² - Fix circleci workspace
- PR #3277²⁸⁷³ - Support external plugins
- PR #3276²⁸⁷⁴ - Fix missing parenthesis in hello_compute.cu.
- PR #3274²⁸⁷⁵ - Reinit counters synchronously in reinit_counters test
- PR #3273²⁸⁷⁶ - Splitting tests to avoid compiler OOM
- PR #3271²⁸⁷⁷ - Remove leftover code from context_generic_context.hpp
- PR #3269²⁸⁷⁸ - Fix bulk_construct with count = 0
- PR #3268²⁸⁷⁹ - Replace constexpr with HPX_CXX14_CONSTEXPR and HPX_CONSTEXPR
- PR #3266²⁸⁸⁰ - Replace boost::format with custom sprintf-based implementation
- PR #3265²⁸⁸¹ - Split parallel tests on CircleCI
- PR #3262²⁸⁸² - Making sure documentation correctly links to source files
- PR #3261²⁸⁸³ - Apex refactoring fix rebind
- PR #3260²⁸⁸⁴ - Isolate performance counter parser into a separate TU
- PR #3256²⁸⁸⁵ - Post 1.1.0 version bumps
- PR #3254²⁸⁸⁶ - Adding trait for actions allowing to make runtime decision on whether to execute it directly
- PR #3253²⁸⁸⁷ - Bump minimal supported Boost to 1.58.0
- PR #3251²⁸⁸⁸ - Adds new feature: changing interval used in interval_timer (issue 3244)
- PR #3239²⁸⁸⁹ - Changing std::rand() to a better inbuilt PRNG generator.
- PR #3234²⁸⁹⁰ - Disable background thread when networking is off

²⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3283>

²⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3282>

²⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3281>

²⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3279>

²⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3278>

²⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3277>

²⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3276>

²⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3274>

²⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3273>

²⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3271>

2878 <https://github.com/STELLAR-GROUP/hpx/pull/3269>2879 <https://github.com/STELLAR-GROUP/hpx/pull/3268>2880 <https://github.com/STELLAR-GROUP/hpx/pull/3266>2881 <https://github.com/STELLAR-GROUP/hpx/pull/3265>2882 <https://github.com/STELLAR-GROUP/hpx/pull/3262>2883 <https://github.com/STELLAR-GROUP/hpx/pull/3261>2884 <https://github.com/STELLAR-GROUP/hpx/pull/3260>2885 <https://github.com/STELLAR-GROUP/hpx/pull/3256>2886 <https://github.com/STELLAR-GROUP/hpx/pull/3254>2887 <https://github.com/STELLAR-GROUP/hpx/pull/3253>2888 <https://github.com/STELLAR-GROUP/hpx/pull/3251>2889 <https://github.com/STELLAR-GROUP/hpx/pull/3239>2890 <https://github.com/STELLAR-GROUP/hpx/pull/3234>

- PR #3232²⁸⁹¹ - Clean up suspension tests
- PR #3230²⁸⁹² - Add optional scheduler mode parameter to create_thread_pool function
- PR #3228²⁸⁹³ - Allow suspension also on static schedulers
- PR #3163²⁸⁹⁴ - libfabric parcelport w/o HPX_PARCELPORT_LIBFABRIC_ENDPOINT_RDM
- PR #3036²⁸⁹⁵ - Switching to CircleCI 2.0

2.10.14 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 OpenMP²⁸⁹⁶ and/or MPI²⁸⁹⁷.
- We have continued to implement various parallel algorithms. *HPX* now almost completely implements all of the parallel algorithms as specified by the C++17 Standard²⁸⁹⁸. We have also continued to implement these algorithms for the distributed use case (for segmented data structures, such as `hpx::partitioned_vector`).
- Added a compatibility layer for `std::thread`, `std::mutex`, and `std::condition_variable` allowing for the code to use those facilities where available and to fall back to the corresponding Boost facilities otherwise. The CMake²⁸⁹⁹ 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 CMake²⁹⁰⁰.
- 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.

²⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3232>

²⁸⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3230>

²⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3228>

²⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3163>

²⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3036>

²⁸⁹⁶ <https://openmp.org/wp/>

²⁸⁹⁷ https://en.wikipedia.org/wiki/Message_Passing_Interface

²⁸⁹⁸ <http://www.open-std.org/jtc1/sc22/wg21>

²⁸⁹⁹ <https://www.cmake.org>

²⁹⁰⁰ <https://www.cmake.org>

- The parallel algorithms adopted for C++17 restrict the iterator categories usable with those to at least forward iterators. Our implementation of the parallel algorithms was supporting input iterators (and output iterators) as well by simply falling back to sequential execution. We have now made our implementations conforming by requiring at least forward iterators. In order to enable the old behavior use the compatibility option `-DHPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT=On` on the [CMake](#)²⁹⁰¹ command line.
- We have added the functionalities allowing for LCOs being implemented using (simple) components. Before LCOs had to always be implemented using managed components.
- User defined components don't have to be default-constructible anymore. Return types from actions don't have to be default-constructible anymore either. Our serialization layer now in general supports non-default-constructible types.
- We have added a new launch policy `hpx::launch::lazy` that allows one to defer the decision on what launch policy to use to the point of execution. This policy is initialized with a function (object) that – when invoked – is expected to produce the desired launch policy.

Breaking changes

- We have dropped support for the gcc compiler version V4.8. The minimal gcc version we now test on is gcc V4.9. The minimally required version of [CMake](#)²⁹⁰² is now V3.3.2.
- We have dropped support for the Visual Studio 2013 compiler version. The minimal Visual Studio version we now test on is Visual Studio 2015.5.
- We have dropped support for the Boost V1.51-V1.54. The minimal version of Boost we now test is Boost V1.55.
- We have dropped support for the `hpx::util::unwrapped` API. `hpx::util::unwrapped` will stay functional to some degree, until it finally gets removed in a later version of HPX. The functional usage of `hpx::util::unwrapped` should be changed to the new `hpx::util::unwrapping` function whereas the immediate usage should be replaced to `hpx::util::unwrap`.
- The performance counter names referring to properties as exposed by the threading subsystem have changes as those now additionally have to specify the thread-pool. See the corresponding documentation for more details.
- The overloads of `hpx::async` that invoke an action do not perform implicit unwrapping of the returned future anymore in case the invoked function does return a future in the first place. In this case `hpx::async` now returns a `hpx::future<future<T>>` making its behavior conforming to its local counterpart.
- We have replaced the use of `boost::exception_ptr` in our APIs with the equivalent `std::exception_ptr`. Please change your codes accordingly. No compatibility settings are provided.
- We have removed the compatibility settings for `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` and `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY` as their life-cycle has reached its end.
- We have removed the experimental thread schedulers `hierarchy_scheduler`, `periodic_priority_scheduler` and `throttling_scheduler` in an effort to clean up and consolidate our thread schedulers.

²⁹⁰¹ <https://www.cmake.org>

²⁹⁰² <https://www.cmake.org>

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #3250²⁹⁰³ - Apex refactoring with guids
- PR #3249²⁹⁰⁴ - Updating People.qbk
- PR #3246²⁹⁰⁵ - Assorted fixes for CUDA
- PR #3245²⁹⁰⁶ - Apex refactoring with guids
- PR #3242²⁹⁰⁷ - Modify task counting in thread_queue.hpp
- PR #3240²⁹⁰⁸ - Fixed typos
- PR #3238²⁹⁰⁹ - Readding accidentally removed std::abort
- PR #3237²⁹¹⁰ - Adding Pipeline example
- PR #3236²⁹¹¹ - Fixing memory_block
- PR #3233²⁹¹² - Make schedule_thread take suspended threads into account
- Issue #3226²⁹¹³ - memory_block is breaking, signaling SIGSEGV on a thread on creation and freeing
- PR #3225²⁹¹⁴ - Applying quick fix for hwloc-2.0
- Issue #3224²⁹¹⁵ - HPX counters crashing the application
- PR #3223²⁹¹⁶ - Fix returns when setting config entries
- Issue #3222²⁹¹⁷ - Errors linking libhpx.so
- Issue #3221²⁹¹⁸ - HPX on Mac OS X with HWLoc 2.0.0 fails to run
- PR #3216²⁹¹⁹ - Reorder a variadic array to satisfy VS 2017 15.6
- PR #3214²⁹²⁰ - Changed prerequisites.qbk to avoid confusion while building boost
- PR #3213²⁹²¹ - Relax locks for thread suspension to avoid holding locks when yielding
- PR #3212²⁹²² - Fix check in sequenced_executor test
- PR #3211²⁹²³ - Use preinit_array to set argc/argv in init_globally example

²⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/3250>

²⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3249>

²⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3246>

²⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3245>

²⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3242>

²⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3240>

²⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3238>

²⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3237>

²⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3236>

²⁹¹² <https://github.com/STELLAR-GROUP/hpx/pull/3233>

²⁹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/3226>

²⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3225>

²⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3224>

²⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3223>

²⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3222>

²⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3221>

²⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3216>

²⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3214>

²⁹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3213>

²⁹²² <https://github.com/STELLAR-GROUP/hpx/pull/3212>

²⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/3211>

- PR #3210²⁹²⁴ - Adapted parallel::{search | search_n} for Ranges TS (see #1668)
- PR #3209²⁹²⁵ - Fix locking problems during shutdown
- Issue #3208²⁹²⁶ - init_globally throwing a run-time error
- PR #3206²⁹²⁷ - Addition of new arithmetic performance counter “Count”
- PR #3205²⁹²⁸ - Fixing return type calculation for bulk_then_execute
- PR #3204²⁹²⁹ - Changing std::rand() to a better inbuilt PRNG generator
- PR #3203²⁹³⁰ - Resolving problems during shutdown for VS2015
- PR #3202²⁹³¹ - Making sure resource partitioner is not accessed if its not valid
- PR #3201²⁹³² - Fixing optional::swap
- Issue #3200²⁹³³ - hpx::util::optional fails
- PR #3199²⁹³⁴ - Fix sliding_semaphore test
- PR #3198²⁹³⁵ - Set pre_main status before launching run_helper
- PR #3197²⁹³⁶ - Update README.rst
- PR #3194²⁹³⁷ - parallel::{fill|fill_n} updated for Ranges TS
- PR #3193²⁹³⁸ - Updating Runtime.cpp by adding correct description of Performance counters during register
- PR #3191²⁹³⁹ - Fix sliding_semaphore_2338 test
- PR #3190²⁹⁴⁰ - Topology improvements
- PR #3189²⁹⁴¹ - Deleting one include of median from BOOST library to arithmetics_counter file
- PR #3188²⁹⁴² - Optionally disable printing of diagnostics during terminate
- PR #3187²⁹⁴³ - Suppressing cmake warning issued by cmake > V3.11
- PR #3185²⁹⁴⁴ - Remove unused scoped_unlock, unlock_guard_try
- PR #3184²⁹⁴⁵ - Fix nqueen example
- PR #3183²⁹⁴⁶ - Add runtime start/stop, resume/suspend and OpenMP benchmarks

²⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3210>

²⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3209>

²⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3208>

²⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3206>

²⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3205>

²⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3204>

²⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3203>

²⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3202>

²⁹³² <https://github.com/STELLAR-GROUP/hpx/pull/3201>

²⁹³³ <https://github.com/STELLAR-GROUP/hpx/issues/3200>

²⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3199>

²⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3198>

²⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3197>

²⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3194>

²⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3193>

²⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3191>

²⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3190>

²⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3189>

²⁹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3188>

²⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3187>

²⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3185>

²⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3184>

²⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3183>

- Issue #3182²⁹⁴⁷ - bulk_then_execute has unexpected return type/does not compile
- Issue #3181²⁹⁴⁸ - hwloc 2.0 breaks topo class and cannot be used
- Issue #3180²⁹⁴⁹ - Schedulers that don't support suspend/resume are unusable
- PR #3179²⁹⁵⁰ - Various minor changes to support FLeCSI
- PR #3178²⁹⁵¹ - Fix #3124
- PR #3177²⁹⁵² - Removed allgather
- PR #3176²⁹⁵³ - Fixed Documentation for “using_hpx_pkgconfig”
- PR #3174²⁹⁵⁴ - Add hpx::iostreams::ostream overload to format_to
- PR #3172²⁹⁵⁵ - Fix lifo queue backend
- PR #3171²⁹⁵⁶ - adding the missing unset() function to cpu_mask() for case of more than 64 threads
- PR #3170²⁹⁵⁷ - Add cmake flag -DHPX_WITHFAULT_TOLERANCE=ON (OFF by default)
- PR #3169²⁹⁵⁸ - Adapted parallel::{count|count_if} for Ranges TS (see #1668)
- PR #3168²⁹⁵⁹ - Changing used namespace for seq execution policy
- Issue #3167²⁹⁶⁰ - Update GSoC projects
- Issue #3166²⁹⁶¹ - Application (Octotiger) gets stuck on hpx::finalize when only using one thread
- Issue #3165²⁹⁶² - Compilation of parallel algorithms with HPX_WITH_DATAPAR is broken
- PR #3164²⁹⁶³ - Fixing component migration
- PR #3162²⁹⁶⁴ - regex_from_pattern: escape regex special characters to avoid misinterpretation
- Issue #3161²⁹⁶⁵ - Building HPX with hwloc 2.0.0 fails
- PR #3160²⁹⁶⁶ - Fixing the handling of quoted command line arguments.
- PR #3158²⁹⁶⁷ - Fixing a race with timed suspension (second attempt)
- PR #3157²⁹⁶⁸ - Revert “Fixing a race with timed suspension”
- PR #3156²⁹⁶⁹ - Fixing serialization of classes with incompatible serialize signature

²⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3182>

²⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3181>

²⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/3180>

²⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3179>

²⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/3178>

²⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3177>

²⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3176>

²⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3174>

²⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3172>

²⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3171>

²⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3170>

²⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3169>

²⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3168>

²⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3167>

²⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/3166>

²⁹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/3165>

²⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3164>

²⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3162>

²⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3161>

²⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3160>

²⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3158>

²⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3157>

²⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3156>

- PR #3154²⁹⁷⁰ - More refactorings based on clang-tidy reports
- PR #3153²⁹⁷¹ - Fixing a race with timed suspension
- PR #3152²⁹⁷² - Documentation for runtime suspension
- PR #3151²⁹⁷³ - Use small_vector only from boost version 1.59 onwards
- PR #3150²⁹⁷⁴ - Avoiding more stack overflows
- PR #3148²⁹⁷⁵ - Refactoring component_base and base_action/transfer_base_action
- PR #3147²⁹⁷⁶ - Move yield_while out of detail namespace and into own file
- PR #3145²⁹⁷⁷ - Remove a leftover of the cxx11 std array cleanup
- PR #3144²⁹⁷⁸ - Minor changes to how actions are executed
- PR #3143²⁹⁷⁹ - Fix stack overhead
- PR #3142²⁹⁸⁰ - Fix typo in config.hpp
- PR #3141²⁹⁸¹ - Fixing small_vector compatibility with older boost version
- PR #3140²⁹⁸² - is_heap_text fix
- Issue #3139²⁹⁸³ - Error in is_heap_tests.hpp
- PR #3138²⁹⁸⁴ - Partially reverting #3126
- PR #3137²⁹⁸⁵ - Suspend speedup
- PR #3136²⁹⁸⁶ - Revert “Fixing #2325”
- PR #3135²⁹⁸⁷ - Improving destruction of threads
- Issue #3134²⁹⁸⁸ - HPX_SERIALIZATION_SPLIT_FREE does not stop compiler from looking for serialize() method
- PR #3133²⁹⁸⁹ - Make hwloc compulsory
- PR #3132²⁹⁹⁰ - Update CXX14 constexpr feature test
- PR #3131²⁹⁹¹ - Fixing #2325
- PR #3130²⁹⁹² - Avoid completion handler allocation

²⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3154>

²⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3153>

²⁹⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3152>

²⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3151>

²⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3150>

²⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3148>

²⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3147>

²⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3145>

²⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3144>

²⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3143>

²⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3142>

²⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3141>

²⁹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/3140>

²⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/3139>

²⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3138>

²⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3137>

²⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3136>

²⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3135>

²⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3134>

²⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3133>

²⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3132>

²⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3131>

²⁹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3130>

- PR #3129²⁹⁹³ - Suspend runtime
- PR #3128²⁹⁹⁴ - Make docbook dtd and xsl path names consistent
- PR #3127²⁹⁹⁵ - Add hpx::start nullptr overloads
- PR #3126²⁹⁹⁶ - Cleaning up coroutine implementation
- PR #3125²⁹⁹⁷ - Replacing nullptr with hpx::threads::invalid_thread_id
- Issue #3124²⁹⁹⁸ - Add hello_world_component to CI builds
- PR #3123²⁹⁹⁹ - Add new constructor.
- PR #3122³⁰⁰⁰ - Fixing #3121
- Issue #3121³⁰⁰¹ - HPX_SMT_PAUSE is broken on non-x86 platforms when __GNUC__ is defined
- PR #3120³⁰⁰² - Don't use boost::intrusive_ptr for thread_id_type
- PR #3119³⁰⁰³ - Disable default executor compatibility with V1 executors
- PR #3118³⁰⁰⁴ - Adding performance_counter::reinit to allow for dynamically changing counter sets
- PR #3117³⁰⁰⁵ - Replace uses of boost/experimental::optional with util::optional
- PR #3116³⁰⁰⁶ - Moving background thread APEX timer #2980
- PR #3115³⁰⁰⁷ - Fixing race condition in channel test
- PR #3114³⁰⁰⁸ - Avoid using util::function for thread function wrappers
- PR #3113³⁰⁰⁹ - cmake V3.10.2 has changed the variable names used for MPI
- PR #3112³⁰¹⁰ - Minor fixes to exclusive_scan algorithm
- PR #3111³⁰¹¹ - Revert "fix detection of cxx11_std_atomic"
- PR #3110³⁰¹² - Suspend thread pool
- PR #3109³⁰¹³ - Fixing thread scheduling when yielding a thread id
- PR #3108³⁰¹⁴ - Revert "Suspend thread pool"
- PR #3107³⁰¹⁵ - Remove UB from thread::id relational operators

²⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3129>

²⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3128>

²⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3127>

²⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3126>

²⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3125>

²⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3124>

²⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3123>

³⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3122>

³⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/3121>

3002 <https://github.com/STELLAR-GROUP/hpx/pull/3120>3003 <https://github.com/STELLAR-GROUP/hpx/pull/3119>3004 <https://github.com/STELLAR-GROUP/hpx/pull/3118>3005 <https://github.com/STELLAR-GROUP/hpx/pull/3117>3006 <https://github.com/STELLAR-GROUP/hpx/pull/3116>3007 <https://github.com/STELLAR-GROUP/hpx/pull/3115>3008 <https://github.com/STELLAR-GROUP/hpx/pull/3114>3009 <https://github.com/STELLAR-GROUP/hpx/pull/3113>3010 <https://github.com/STELLAR-GROUP/hpx/pull/3112>3011 <https://github.com/STELLAR-GROUP/hpx/pull/3111>3012 <https://github.com/STELLAR-GROUP/hpx/pull/3110>3013 <https://github.com/STELLAR-GROUP/hpx/pull/3109>3014 <https://github.com/STELLAR-GROUP/hpx/pull/3108>3015 <https://github.com/STELLAR-GROUP/hpx/pull/3107>

- PR #3106³⁰¹⁶ - Add cmake test for std::decay_t to fix cuda build
- PR #3105³⁰¹⁷ - Fixing refcount for async traversal frame
- PR #3104³⁰¹⁸ - Local execution of direct actions is now actually performed directly
- PR #3103³⁰¹⁹ - Adding support for generic counter_raw_values performance counter type
- Issue #3102³⁰²⁰ - Introduce generic performance counter type returning an array of values
- PR #3101³⁰²¹ - Revert “Adapting stack overhead limit for gcc 4.9”
- PR #3100³⁰²² - Fix #3068 (condition_variable deadlock)
- PR #3099³⁰²³ - Fixing lock held during suspension in papi counter component
- PR #3098³⁰²⁴ - Unbreak broadcast_wait_for_2822 test
- PR #3097³⁰²⁵ - Adapting stack overhead limit for gcc 4.9
- PR #3096³⁰²⁶ - fix detection of cxx11_std_atomic
- PR #3095³⁰²⁷ - Add ciso646 header to get _LIBCPP_VERSION for testing inplace merge
- PR #3094³⁰²⁸ - Relax atomic operations on performance counter values
- PR #3093³⁰²⁹ - Short-circuit all_of/any_of/none_of instantiations
- PR #3092³⁰³⁰ - Take advantage of C++14 lambda capture initialization syntax, where possible
- PR #3091³⁰³¹ - Remove more references to Boost from logging code
- PR #3090³⁰³² - Unify use of yield/yield_k
- PR #3089³⁰³³ - Fix a strange thing in parallel::detail::handle_exception. (Fix #2834.)
- Issue #3088³⁰³⁴ - A strange thing in parallel::sort.
- PR #3087³⁰³⁵ - Fixing assertion in default_distribution_policy
- PR #3086³⁰³⁶ - Implement parallel::remove and parallel::remove_if
- PR #3085³⁰³⁷ - Addressing breaking changes in Boost V1.66
- PR #3084³⁰³⁸ - Ignore build warnings round 2

³⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3106>

³⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3105>

³⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3104>

³⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3103>

³⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3102>

³⁰²¹ <https://github.com/STELLAR-GROUP/hpx/pull/3101>

³⁰²² <https://github.com/STELLAR-GROUP/hpx/pull/3100>

³⁰²³ <https://github.com/STELLAR-GROUP/hpx/pull/3099>

³⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3098>

³⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3097>

³⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3096>

³⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3095>

³⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3094>

³⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3093>

³⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3092>

³⁰³¹ <https://github.com/STELLAR-GROUP/hpx/pull/3091>

³⁰³² <https://github.com/STELLAR-GROUP/hpx/pull/3090>

³⁰³³ <https://github.com/STELLAR-GROUP/hpx/pull/3089>

³⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3088>

³⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3087>

³⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3086>

³⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3085>

³⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3084>

- PR #3083³⁰³⁹ - Fix typo HPX_WITH_MM_PREFECTH
- PR #3081³⁰⁴⁰ - Pre-decay template arguments early
- PR #3080³⁰⁴¹ - Suspend thread pool
- PR #3079³⁰⁴² - Ignore build warnings
- PR #3078³⁰⁴³ - Don't test inplace_merge with libc++
- PR #3076³⁰⁴⁴ - Fixing 3075: Part 1
- PR #3074³⁰⁴⁵ - Fix more build warnings
- PR #3073³⁰⁴⁶ - Suspend thread cleanup
- PR #3072³⁰⁴⁷ - Change existing symbol_namespace::iterate to return all data instead of invoking a callback
- PR #3071³⁰⁴⁸ - Fixing pack_traversal_async test
- PR #3070³⁰⁴⁹ - Fix dynamic_counters_loaded_1508 test by adding dependency to memory_component
- PR #3069³⁰⁵⁰ - Fix scheduling loop exit
- Issue #3068³⁰⁵¹ - hpx::lcos::condition_variable could be suspect to deadlocks
- PR #3067³⁰⁵² - #ifdef out random_shuffle deprecated in later c++
- PR #3066³⁰⁵³ - Make coalescing test depend on coalescing library to ensure it gets built
- PR #3065³⁰⁵⁴ - Workaround for minimal_timed_async_executor_test compilation failures, attempts to copy a deferred call (in unevaluated context)
- PR #3064³⁰⁵⁵ - Fixing wrong condition in wrapper_heap
- PR #3062³⁰⁵⁶ - Fix exception handling for execution::seq
- PR #3061³⁰⁵⁷ - Adapt MSVC C++ mode handling to VS15.5
- PR #3060³⁰⁵⁸ - Fix compiler problem in MSVC release mode
- PR #3059³⁰⁵⁹ - Fixing #2931
- Issue #3058³⁰⁶⁰ - minimal_timed_async_executor_test_exe fails to compile on master (d6f505c)
- PR #3057³⁰⁶¹ - Fix stable_merge_2964 compilation problems

³⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3083>

³⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3081>

³⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/3080>

³⁰⁴² <https://github.com/STELLAR-GROUP/hpx/pull/3079>

³⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/3078>

³⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3076>

³⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3074>

³⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3073>

³⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3072>

³⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3071>

³⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3070>

³⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3069>

³⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/3068>

³⁰⁵² <https://github.com/STELLAR-GROUP/hpx/pull/3067>

³⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/3066>

³⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3065>

³⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3064>

³⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3062>

³⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3061>

³⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3060>

³⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3059>

³⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/3058>

³⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/3057>

- PR #3056³⁰⁶² - Fix some build warnings caused by unused variables/unnecessary tests
- PR #3055³⁰⁶³ - Update documentation for running tests
- Issue #3054³⁰⁶⁴ - Assertion failure when using bulk hpx::new_ in asynchronous mode
- PR #3052³⁰⁶⁵ - Do not bind test running to cmake test build rule
- PR #3051³⁰⁶⁶ - Fix HPX-Qt interaction in Qt example.
- Issue #3048³⁰⁶⁷ - nqueen example fails occasionally
- PR #3047³⁰⁶⁸ - Fixing #3044
- PR #3046³⁰⁶⁹ - Add OS thread suspension
- PR #3042³⁰⁷⁰ - PyCicle - first attempt at a build tool for checking PR's
- PR #3041³⁰⁷¹ - Fix a problem about asynchronous execution of parallel::merge and parallel::partition.
- PR #3040³⁰⁷² - Fix a mistake about exception handling in asynchronous execution of scan_partitioner.
- PR #3039³⁰⁷³ - Consistently use executors to schedule work
- PR #3038³⁰⁷⁴ - Fixing local direct function execution and lambda actions perfect forwarding
- PR #3035³⁰⁷⁵ - Make parallel unit test names match build target/folder names
- PR #3033³⁰⁷⁶ - Fix setting of default build type
- Issue #3032³⁰⁷⁷ - Fix partitioner arg copy found in #2982
- Issue #3031³⁰⁷⁸ - Errors linking libhpx.so due to missing references (master branch, commit 6679a8882)
- PR #3030³⁰⁷⁹ - Revert "implement executor then interface with && forwarding reference"
- PR #3029³⁰⁸⁰ - Run CI inspect checks before building
- PR #3028³⁰⁸¹ - Added range version of parallel::move
- Issue #3027³⁰⁸² - Implement all scheduling APIs in terms of executors
- PR #3026³⁰⁸³ - implement executor then interface with && forwarding reference
- PR #3025³⁰⁸⁴ - Fix typo uninitialized to uninitialized

³⁰⁶² <https://github.com/STELLAR-GROUP/hpx/pull/3056>

³⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/3055>

³⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/3054>

³⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3052>

³⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3051>

³⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3048>

³⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3047>

³⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3046>

³⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3042>

³⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/3041>

³⁰⁷² <https://github.com/STELLAR-GROUP/hpx/pull/3040>

³⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/3039>

³⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3038>

³⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3035>

³⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3033>

³⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/3032>

³⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/3031>

³⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3030>

³⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3029>

³⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/3028>

³⁰⁸² <https://github.com/STELLAR-GROUP/hpx/issues/3027>

³⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/3026>

³⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3025>

- PR #3024³⁰⁸⁵ - Inspect fixes
- PR #3023³⁰⁸⁶ - P0356 Simplified partial function application
- PR #3022³⁰⁸⁷ - Master fixes
- PR #3021³⁰⁸⁸ - Segfault fix
- PR #3020³⁰⁸⁹ - Disable command-line aliasing for applications that use user_main
- PR #3019³⁰⁹⁰ - Adding enable_elasticity option to pool configuration
- PR #3018³⁰⁹¹ - Fix stack overflow detection configuration in header files
- PR #3017³⁰⁹² - Speed up local action execution
- PR #3016³⁰⁹³ - Unify stack-overflow detection options, remove reference to libsigsegv
- PR #3015³⁰⁹⁴ - Speeding up accessing the resource partitioner and the topology info
- Issue #3014³⁰⁹⁵ - HPX does not compile on POWER8 with gcc 5.4
- Issue #3013³⁰⁹⁶ - hello_world occasionally prints multiple lines from a single OS-thread
- PR #3012³⁰⁹⁷ - Silence warning about casting away qualifiers in itt_notify.hpp
- PR #3011³⁰⁹⁸ - Fix cpuset leak in hwloc_topology_info.cpp
- PR #3010³⁰⁹⁹ - Remove useless decay_copy
- PR #3009³¹⁰⁰ - Fixing 2996
- PR #3008³¹⁰¹ - Remove unused internal function
- PR #3007³¹⁰² - Fixing wrapper_heap alignment problems
- Issue #3006³¹⁰³ - hwloc memory leak
- PR #3004³¹⁰⁴ - Silence C4251 (needs to have dll-interface) for future_data_void
- Issue #3003³¹⁰⁵ - Suspension of runtime
- PR #3001³¹⁰⁶ - Attempting to avoid data races in async_traversal while evaluating dataflow()
- PR #3000³¹⁰⁷ - Adding hpx::util::optional as a first step to replace experimental::optional

³⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/3024>

³⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3023>

³⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3022>

³⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3021>

³⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3020>

³⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3019>

³⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/3018>

³⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/3017>

³⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/3016>

³⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3015>

³⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3014>

³⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/3013>

³⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3012>

³⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/3011>

³⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/3010>

³¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/3009>

³¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/3008>

³¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/3007>

³¹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/3006>

³¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/3004>

³¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/3003>

³¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/3001>

³¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/3000>

- PR #2998³¹⁰⁸ - Cleanup up and Fixing component creation and deletion
- Issue #2996³¹⁰⁹ - Build fails with HPX_WITH_HWLOC=OFF
- PR #2995³¹¹⁰ - Push more future_data functionality to source file
- PR #2994³¹¹¹ - WIP: Fix throttle test
- PR #2993³¹¹² - Making sure -hpx:help does not throw for required (but missing) arguments
- PR #2992³¹¹³ - Adding non-blocking (on destruction) service executors
- Issue #2991³¹¹⁴ - run_as_os_thread locks up
- Issue #2990³¹¹⁵ - --help will not work until all required options are provided
- PR #2989³¹¹⁶ - Improve error messages caused by misuse of dataflow
- PR #2988³¹¹⁷ - Improve error messages caused by misuse of .then
- Issue #2987³¹¹⁸ - stack overflow detection producing false positives
- PR #2986³¹¹⁹ - Deduplicate non-dependent thread_info logging types
- PR #2985³¹²⁰ - Adapted parallel::{all_of|any_of|none_of} for Ranges TS (see #1668)
- PR #2984³¹²¹ - Refactor one_size_heap code to simplify code
- PR #2983³¹²² - Fixing local_new_component
- PR #2982³¹²³ - Clang tidy
- PR #2981³¹²⁴ - Simplify allocator rebinding in pack traversal
- PR #2979³¹²⁵ - Fixing integer overflows
- PR #2978³¹²⁶ - Implement parallel::inplace_merge
- Issue #2977³¹²⁷ - Make hwloc compulsory instead of optional
- PR #2976³¹²⁸ - Making sure client_base instance that registered the component does not unregister it when being destructed
- PR #2975³¹²⁹ - Change version of pulled APEX to master
- PR #2974³¹³⁰ - Fix domain not being freed at the end of scheduling loop

³¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2998>

³¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2996>

³¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2995>

³¹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2994>

³¹¹² <https://github.com/STELLAR-GROUP/hpx/pull/2993>

³¹¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2992>

³¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2991>

³¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2990>

³¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2989>

³¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2988>

³¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2987>

³¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2986>

³¹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2985>

³¹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2984>

³¹²² <https://github.com/STELLAR-GROUP/hpx/pull/2983>

³¹²³ <https://github.com/STELLAR-GROUP/hpx/pull/2982>

³¹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2981>

³¹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2979>

³¹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2978>

³¹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2977>

³¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2976>

³¹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2975>

³¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2974>

- PR #2973³¹³¹ - Fix small typos
- PR #2972³¹³² - Adding uintstd.h header
- PR #2971³¹³³ - Fall back to creating local components using local_new
- PR #2970³¹³⁴ - Improve is_tuple_like trait
- PR #2969³¹³⁵ - Fix HPX_WITH_MORE_THAN_64_THREADS default value
- PR #2968³¹³⁶ - Cleaning up dataflow overload set
- PR #2967³¹³⁷ - Make parallel::merge is stable. (Fix #2964.)
- PR #2966³¹³⁸ - Fixing a couple of held locks during exception handling
- PR #2965³¹³⁹ - Adding missing #include
- Issue #2964³¹⁴⁰ - parallel merge is not stable
- PR #2963³¹⁴¹ - Making sure any function object passed to dataflow is released after being invoked
- PR #2962³¹⁴² - Partially reverting #2891
- PR #2961³¹⁴³ - Attempt to fix the gcc 4.9 problem with the async pack traversal
- Issue #2959³¹⁴⁴ - Program terminates during error handling
- Issue #2958³¹⁴⁵ - HPX_PLAIN_ACTION breaks due to missing include
- PR #2957³¹⁴⁶ - Fixing errors generated by mixing different attribute syntaxes
- Issue #2956³¹⁴⁷ - Mixing attribute syntaxes leads to compiler errors
- Issue #2955³¹⁴⁸ - Fix OS-Thread throttling
- PR #2953³¹⁴⁹ - Making sure any hpx.os_threads=N supplied through a -hpx::config file is taken into account
- PR #2952³¹⁵⁰ - Removing wrong call to cleanup_terminated_locked
- PR #2951³¹⁵¹ - Revert “Make sure the function vtables are initialized before use”
- PR #2950³¹⁵² - Fix a namespace compilation error when some schedulers are disabled
- Issue #2949³¹⁵³ - master branch giving lockups on shutdown

³¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2973>

³¹³² <https://github.com/STELLAR-GROUP/hpx/pull/2972>

³¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/2971>

³¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2970>

³¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2969>

³¹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2968>

³¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2967>

³¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2966>

³¹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2965>

³¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2964>

³¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2963>

³¹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2962>

³¹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2961>

³¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2959>

³¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2958>

³¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2957>

³¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2956>

³¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2955>

³¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2953>

³¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2952>

³¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2951>

³¹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2950>

³¹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/2949>

- Issue #2947³¹⁵⁴ - hpx.ini is not used correctly at initialization
- PR #2946³¹⁵⁵ - Adding explicit feature test for thread_local
- PR #2945³¹⁵⁶ - Make sure the function vtables are initialized before use
- PR #2944³¹⁵⁷ - Attempting to solve affinity problems on CircleCI
- PR #2943³¹⁵⁸ - Changing channel actions to be direct
- PR #2942³¹⁵⁹ - Adding split_future for std::vector
- PR #2941³¹⁶⁰ - Add a feature test to test for CXX11 override
- Issue #2940³¹⁶¹ - Add split_future for future<vector<T>>
- PR #2939³¹⁶² - Making error reporting during problems with setting affinity masks more verbose
- PR #2938³¹⁶³ - Fix this various executors
- PR #2937³¹⁶⁴ - Fix some typos in documentation
- PR #2934³¹⁶⁵ - Remove the need for “complete” SFINAE checks
- PR #2933³¹⁶⁶ - Making sure parallel::for_loop is executed in parallel if requested
- PR #2932³¹⁶⁷ - Classify chunk_size_iterator to input iterator tag. (Fix #2866)
- Issue #2931³¹⁶⁸ - –hpx:help triggers unusual error with clang build
- PR #2930³¹⁶⁹ - Add #include files needed to set _POSIX_VERSION for debug check
- PR #2929³¹⁷⁰ - Fix a couple of deprecated c++ features
- PR #2928³¹⁷¹ - Fixing execution parameters
- Issue #2927³¹⁷² - CMake warning: ... cycle in constraint graph
- PR #2926³¹⁷³ - Default pool rename
- Issue #2925³¹⁷⁴ - Default pool cannot be renamed
- Issue #2924³¹⁷⁵ - hpx:attach-debugger=startup does not work any more
- PR #2923³¹⁷⁶ - Alloc membind

³¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2947>

³¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2946>

³¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2945>

³¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2944>

³¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2943>

³¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2942>

³¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2941>

³¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2940>

³¹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2939>

³¹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2938>

³¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2937>

³¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2934>

³¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2933>

³¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2932>

³¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2931>

³¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2930>

³¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2929>

³¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2928>

³¹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/2927>

³¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2926>

³¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2925>

³¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2924>

³¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2923>

- PR #2922³¹⁷⁷ - This fixes CircleCI errors when running with `-hpx:bind=none`
- PR #2921³¹⁷⁸ - Custom pool executor was missing priority and stacksize options
- PR #2920³¹⁷⁹ - Adding test to trigger problem reported in #2916
- PR #2919³¹⁸⁰ - Make sure the resource_partitioner is properly destructed on `hpx::finalize`
- Issue #2918³¹⁸¹ - `hpx::init` calls wrong (first) callback when called multiple times
- PR #2917³¹⁸² - Adding util::checkpoint
- Issue #2916³¹⁸³ - Weird runtime failures when using a channel and chained continuations
- PR #2915³¹⁸⁴ - Introduce executor parameters customization points
- Issue #2914³¹⁸⁵ - Task assignment to current Pool has unintended consequences
- PR #2913³¹⁸⁶ - Fix rp hang
- PR #2912³¹⁸⁷ - Update contributors
- PR #2911³¹⁸⁸ - Fixing CUDA problems
- PR #2910³¹⁸⁹ - Improve error reporting for process component on POSIX systems
- PR #2909³¹⁹⁰ - Fix typo in include path
- PR #2908³¹⁹¹ - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2907³¹⁹² - Optionally force-delete remaining channel items on close
- PR #2906³¹⁹³ - Making sure generated performance counter names are correct
- Issue #2905³¹⁹⁴ - collecting idle-rate performance counters on multiple localities produces an error
- Issue #2904³¹⁹⁵ - build broken for Intel 17 compilers
- PR #2903³¹⁹⁶ - Documentation Updates—Adding New People
- PR #2902³¹⁹⁷ - Fixing service_executor
- PR #2901³¹⁹⁸ - Fixing partitioned_vector creation
- PR #2900³¹⁹⁹ - Add numa-balanced mode to `hpx::bind`, spread cores over numa domains

³¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2922>

³¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2921>

³¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2920>

³¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2919>

³¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/2918>

³¹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2917>

³¹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/2916>

³¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2915>

³¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2914>

3186 <https://github.com/STELLAR-GROUP/hpx/pull/2913>3187 <https://github.com/STELLAR-GROUP/hpx/pull/2912>3188 <https://github.com/STELLAR-GROUP/hpx/pull/2911>3189 <https://github.com/STELLAR-GROUP/hpx/pull/2910>3190 <https://github.com/STELLAR-GROUP/hpx/pull/2909>3191 <https://github.com/STELLAR-GROUP/hpx/pull/2908>3192 <https://github.com/STELLAR-GROUP/hpx/pull/2907>3193 <https://github.com/STELLAR-GROUP/hpx/pull/2906>3194 <https://github.com/STELLAR-GROUP/hpx/issues/2905>3195 <https://github.com/STELLAR-GROUP/hpx/issues/2904>3196 <https://github.com/STELLAR-GROUP/hpx/pull/2903>3197 <https://github.com/STELLAR-GROUP/hpx/pull/2902>3198 <https://github.com/STELLAR-GROUP/hpx/pull/2901>3199 <https://github.com/STELLAR-GROUP/hpx/pull/2900>

- Issue #2899³²⁰⁰ - hpx::bind does not have a mode that balances cores over numa domains
- PR #2898³²⁰¹ - Adding missing #include and missing guard for optional code section
- PR #2897³²⁰² - Removing dependency on Boost.ILC
- Issue #2896³²⁰³ - Debug build fails without -fpermissive with GCC 7.1 and Boost 1.65
- PR #2895³²⁰⁴ - Fixing SLURM environment parsing
- PR #2894³²⁰⁵ - Fix incorrect handling of compile definition with value 0
- Issue #2893³²⁰⁶ - Disabling schedulers causes build errors
- PR #2892³²⁰⁷ - added list serializer
- PR #2891³²⁰⁸ - Resource Partitioner Fixes
- Issue #2890³²⁰⁹ - Destroying a non-empty channel causes an assertion failure
- PR #2889³²¹⁰ - Add check for libatomic
- PR #2888³²¹¹ - Fix compilation problems if HPX_WITH_ITT_NOTIFY=ON
- PR #2887³²¹² - Adapt broadcast() to non-unwrapping async<Action>
- PR #2886³²¹³ - Replace Boost.Random with C++11 <random>
- Issue #2885³²¹⁴ - regression in broadcast?
- Issue #2884³²¹⁵ - linking -latomic is not portable
- PR #2883³²¹⁶ - Explicitly set -pthread flag if available
- PR #2882³²¹⁷ - Wrap boost::format uses
- Issue #2881³²¹⁸ - hpx not compiling with HPX_WITH_ITTNOTIFY=On
- Issue #2880³²¹⁹ - hpx::bind scatter/balanced give wrong pu masks
- PR #2878³²²⁰ - Fix incorrect pool usage masks setup in RP/thread manager
- PR #2877³²²¹ - Require std::array by default
- PR #2875³²²² - Deprecate use of BOOST_ASSERT

³²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2899>

³²⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2898>

³²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2897>

³²⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2896>

³²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2895>

³²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2894>

³²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2893>

³²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2892>

³²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2891>

³²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2890>

³²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2889>

³²¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2888>

³²¹² <https://github.com/STELLAR-GROUP/hpx/pull/2887>

³²¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2886>

³²¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2885>

³²¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2884>

³²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2883>

³²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2882>

³²¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2881>

³²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2880>

³²²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2878>

³²²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2877>

³²²² <https://github.com/STELLAR-GROUP/hpx/pull/2875>

- PR #2874³²²³ - Changed serialization of boost.variant to use variadic templates
- Issue #2873³²²⁴ - building with parcelport_mpi fails on cori
- PR #2871³²²⁵ - Adding missing support for throttling scheduler
- PR #2870³²²⁶ - Disambiguate use of base_lco_with_value macros with channel
- Issue #2869³²²⁷ - Difficulty compiling HPX_REGISTER_CHANNEL_DECLARATION(double)
- PR #2868³²²⁸ - Removing unneeded assert
- PR #2867³²²⁹ - Implement parallel::unique
- Issue #2866³²³⁰ - The chunk_size_iterator violates multipass guarantee
- PR #2865³²³¹ - Only use sched_getcpu on linux machines
- PR #2864³²³² - Create redistribution archive for successful builds
- PR #2863³²³³ - Replace casts/assignments with hard-coded memcpy operations
- Issue #2862³²³⁴ - sched_getcpu not available on MacOS
- PR #2861³²³⁵ - Fixing unmatched header defines and recursive inclusion of threadmanager
- Issue #2860³²³⁶ - Master program fails with assertion ‘type == data_type_address’ failed: HPX(assertion_failure)
- Issue #2852³²³⁷ - Support for ARM64
- PR #2858³²³⁸ - Fix misplaced #if #endif’s that cause build failure without THREAD_CUMULATIVE_COUNTS
- PR #2857³²³⁹ - Fix some listing in documentation
- PR #2856³²⁴⁰ - Fixing component handling for lcos
- PR #2855³²⁴¹ - Add documentation for coarrays
- PR #2854³²⁴² - Support ARM64 in timestamps
- PR #2853³²⁴³ - Update Table 17. Non-modifying Parallel Algorithms in Documentation
- PR #2851³²⁴⁴ - Allowing for non-default-constructible component types
- PR #2850³²⁴⁵ - Enable returning future<R> from actions where R is not default-constructible

³²²³ <https://github.com/STELLAR-GROUP/hpx/pull/2874>

³²²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2873>

³²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2871>

³²²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2870>

³²²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2869>

³²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2868>

³²²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2867>

³²³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2866>

³²³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2865>

3232 <https://github.com/STELLAR-GROUP/hpx/pull/2864>3233 <https://github.com/STELLAR-GROUP/hpx/pull/2863>3234 <https://github.com/STELLAR-GROUP/hpx/issues/2862>3235 <https://github.com/STELLAR-GROUP/hpx/pull/2861>3236 <https://github.com/STELLAR-GROUP/hpx/issues/2860>3237 <https://github.com/STELLAR-GROUP/hpx/issues/2852>3238 <https://github.com/STELLAR-GROUP/hpx/pull/2858>3239 <https://github.com/STELLAR-GROUP/hpx/pull/2857>3240 <https://github.com/STELLAR-GROUP/hpx/pull/2856>3241 <https://github.com/STELLAR-GROUP/hpx/pull/2855>3242 <https://github.com/STELLAR-GROUP/hpx/pull/2854>3243 <https://github.com/STELLAR-GROUP/hpx/pull/2853>3244 <https://github.com/STELLAR-GROUP/hpx/pull/2851>3245 <https://github.com/STELLAR-GROUP/hpx/pull/2850>

- PR #2849³²⁴⁶ - Unify serialization of non-default-constructable types
- Issue #2848³²⁴⁷ - Components have to be default constructible
- Issue #2847³²⁴⁸ - Returning a future<R> where R is not default-constructable broken
- Issue #2846³²⁴⁹ - Unify serialization of non-default-constructible types
- PR #2845³²⁵⁰ - Add Visual Studio 2015 to the tested toolchains in Appveyor
- Issue #2844³²⁵¹ - Change the appveyor build to use the minimal required MSVC version
- Issue #2843³²⁵² - multi node hello_world hangs
- PR #2842³²⁵³ - Correcting Spelling mistake in docs
- PR #2841³²⁵⁴ - Fix usage of std::aligned_storage
- PR #2840³²⁵⁵ - Remove constexpr from a void function
- Issue #2839³²⁵⁶ - memcpy buffer overflow: load_construct_data() and std::complex members
- Issue #2835³²⁵⁷ - constexpr functions with void return type break compilation with CUDA 8.0
- Issue #2834³²⁵⁸ - One suspicion in parallel::detail::handle_exception
- PR #2833³²⁵⁹ - Implement parallel::merge
- PR #2832³²⁶⁰ - Fix a strange thing in parallel::util::detail::handle_local_exceptions. (Fix #2818)
- PR #2830³²⁶¹ - Break the debugger when a test failed
- Issue #2831³²⁶² - parallel/executors/execution_fwd.hpp causes compilation failure in C++11 mode.
- PR #2829³²⁶³ - Implement an API for asynchronous pack traversal
- PR #2828³²⁶⁴ - Split unit test builds on CircleCI to avoid timeouts
- Issue #2827³²⁶⁵ - failure to compile hello_world example with -Werror
- PR #2824³²⁶⁶ - Making sure promises are marked as started when used as continuations
- PR #2823³²⁶⁷ - Add documentation for partitioned_vector_view
- Issue #2822³²⁶⁸ - Yet another issue with wait_for similar to #2796

³²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2849>

³²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2848>

³²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2847>

³²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2846>

³²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2845>

³²⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2844>

³²⁵² <https://github.com/STELLAR-GROUP/hpx/issues/2843>

³²⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2842>

³²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2841>

³²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2840>

³²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2839>

³²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2835>

³²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2834>

³²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2833>

³²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2832>

³²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2830>

³²⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2831>

³²⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2829>

³²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2828>

³²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2827>

³²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2824>

³²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2823>

³²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2822>

- PR #2821³²⁶⁹ - Fix bugs and improve that about HPX_HAVE_CXX11_AUTO_RETURN_VALUE of CMake
- PR #2820³²⁷⁰ - Support C++11 in benchmark codes of parallel::partition and parallel::partition_copy
- PR #2819³²⁷¹ - Fix compile errors in unit test of container version of parallel::partition
- Issue #2818³²⁷² - A strange thing in parallel::util::detail::handle_local_exceptions
- Issue #2815³²⁷³ - HPX fails to compile with HPX_WITH_CUDA=ON and the new CUDA 9.0 RC
- Issue #2814³²⁷⁴ - Using ‘gmakeN’ after ‘cmake’ produces error in src/CMakeFiles/hpx.dir/runtime/agas/addressing_service.cpp.o
- PR #2813³²⁷⁵ - Properly support [[noreturn]] attribute if available
- Issue #2812³²⁷⁶ - Compilation fails with gcc 7.1.1
- PR #2811³²⁷⁷ - Adding hpx::launch::lazy and support for async, dataflow, and future::then
- PR #2810³²⁷⁸ - Add option allowing to disable deprecation warning
- PR #2809³²⁷⁹ - Disable throttling scheduler if HWLOC is not found/used
- PR #2808³²⁸⁰ - Fix compile errors on some environments of parallel::partition
- Issue #2807³²⁸¹ - Difficulty building with HPX_WITH_HWLOC=Off
- PR #2806³²⁸² - Partitioned vector
- PR #2805³²⁸³ - Serializing collections with non-default constructible data
- PR #2802³²⁸⁴ - Fix FreeBSD 11
- Issue #2801³²⁸⁵ - Rate limiting techniques in io_service
- Issue #2800³²⁸⁶ - New Launch Policy: async_if
- PR #2799³²⁸⁷ - Fix a unit test failure on GCC in tuple_cat
- PR #2798³²⁸⁸ - bump minimum required cmake to 3.0 in test
- PR #2797³²⁸⁹ - Making sure future::wait_for et.al. work properly for action results
- Issue #2796³²⁹⁰ - wait_for does always in “deferred” state for calls on remote localities
- Issue #2795³²⁹¹ - Serialization of types without default constructor

³²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2821>

³²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2820>

³²⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2819>

³²⁷² <https://github.com/STELLAR-GROUP/hpx/issues/2818>

³²⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/2815>

³²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2814>

³²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2813>

³²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2812>

³²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2811>

3278 <https://github.com/STELLAR-GROUP/hpx/pull/2810>3279 <https://github.com/STELLAR-GROUP/hpx/pull/2809>3280 <https://github.com/STELLAR-GROUP/hpx/pull/2808>3281 <https://github.com/STELLAR-GROUP/hpx/issues/2807>3282 <https://github.com/STELLAR-GROUP/hpx/pull/2806>3283 <https://github.com/STELLAR-GROUP/hpx/pull/2805>3284 <https://github.com/STELLAR-GROUP/hpx/pull/2802>3285 <https://github.com/STELLAR-GROUP/hpx/issues/2801>3286 <https://github.com/STELLAR-GROUP/hpx/issues/2800>3287 <https://github.com/STELLAR-GROUP/hpx/pull/2799>3288 <https://github.com/STELLAR-GROUP/hpx/pull/2798>3289 <https://github.com/STELLAR-GROUP/hpx/pull/2797>3290 <https://github.com/STELLAR-GROUP/hpx/issues/2796>3291 <https://github.com/STELLAR-GROUP/hpx/issues/2795>

- PR #2794³²⁹² - Fixing test for partitioned_vector iteration
- PR #2792³²⁹³ - Implemented segmented find and its variations for partitioned vector
- PR #2791³²⁹⁴ - Circumvent scary warning about placement new
- PR #2790³²⁹⁵ - Fix OSX build
- PR #2789³²⁹⁶ - Resource partitioner
- PR #2788³²⁹⁷ - Adapt parallel::is_heap and parallel::is_heap_until to Ranges TS
- PR #2787³²⁹⁸ - Unwrap hotfixes
- PR #2786³²⁹⁹ - Update CMake Minimum Version to 3.3.2 (refs #2565)
- Issue #2785³³⁰⁰ - Issues with masks and cpuset
- PR #2784³³⁰¹ - Error with reduce and transform reduce fixed
- PR #2783³³⁰² - StackOverflow integration with libsigsegv
- PR #2782³³⁰³ - Replace boost::atomic with std::atomic (where possible)
- PR #2781³³⁰⁴ - Check for and optionally use [[deprecated]] attribute
- PR #2780³³⁰⁵ - Adding empty (but non-trivial) destructor to circumvent warnings
- PR #2779³³⁰⁶ - Exception info tweaks
- PR #2778³³⁰⁷ - Implement parallel::partition
- PR #2777³³⁰⁸ - Improve error handling in gather_here/gather_there
- PR #2776³³⁰⁹ - Fix a bug in compiler version check
- PR #2775³³¹⁰ - Fix compilation when HPX_WITH_LOGGING is OFF
- PR #2774³³¹¹ - Removing dependency on Boost.Date_Time
- PR #2773³³¹² - Add sync_images() method to spmd_block class
- PR #2772³³¹³ - Adding documentation for PAPI counters
- PR #2771³³¹⁴ - Removing boost preprocessor dependency

³²⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2794>

³²⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2792>

³²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2791>

³²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2790>

³²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2789>

³²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2788>

³²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2787>

³²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2786>

³³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2785>

³³⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2784>

³³⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2783>

³³⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2782>

³³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2781>

³³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2780>

³³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2779>

³³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2778>

³³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2777>

³³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2776>

³³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2775>

³³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2774>

³³¹² <https://github.com/STELLAR-GROUP/hpx/pull/2773>

³³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2772>

³³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2771>

- PR #2770³³¹⁵ - Adding test, fixing deadlock in config registry
- PR #2769³³¹⁶ - Remove some other warnings and errors detected by clang 5.0
- Issue #2768³³¹⁷ - Is there iterator tag for HPX?
- PR #2767³³¹⁸ - Improvements to continuation annotation
- PR #2765³³¹⁹ - gcc split stack support for HPX threads #620
- PR #2764³³²⁰ - Fix some uses of begin/end, remove unnecessary includes
- PR #2763³³²¹ - Bump minimal Boost version to 1.55.0
- PR #2762³³²² - hpx::partitioned_vector serializer
- PR #2761³³²³ - Adding configuration summary to cmake output and –hpx:info
- PR #2760³³²⁴ - Removing 1d_hydro example as it is broken
- PR #2758³³²⁵ - Remove various warnings detected by clang 5.0
- Issue #2757³³²⁶ - In case of a “raw thread” is needed per core for implementing parallel algorithm, what is good practice in HPX?
- PR #2756³³²⁷ - Allowing for LCOs to be simple components
- PR #2755³³²⁸ - Removing make_index_pack_unrolled
- PR #2754³³²⁹ - Implement parallel::unique_copy
- PR #2753³³³⁰ - Fixing detection of [[fallthrough]] attribute
- PR #2752³³³¹ - New thread priority names
- PR #2751³³³² - Replace boost::exception with proposed exception_info
- PR #2750³³³³ - Replace boost::iterator_range
- PR #2749³³³⁴ - Fixing hdf5 examples
- Issue #2748³³³⁵ - HPX fails to build with enabled hdf5 examples
- Issue #2747³³³⁶ - Inherited task priorities break certain DAG optimizations
- Issue #2746³³³⁷ - HPX segfaulting with valgrind

³³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2770>

³³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2769>

³³¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2768>

³³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2767>

³³¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2765>

³³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2764>

³³²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2763>

³³²² <https://github.com/STELLAR-GROUP/hpx/pull/2762>

³³²³ <https://github.com/STELLAR-GROUP/hpx/pull/2761>

³³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2760>

³³²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2758>

³³²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2757>

³³²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2756>

³³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2755>

³³²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2754>

³³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2753>

³³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2752>

³³³² <https://github.com/STELLAR-GROUP/hpx/pull/2751>

³³³³ <https://github.com/STELLAR-GROUP/hpx/pull/2750>

³³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2749>

³³³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2748>

³³³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2747>

³³³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2746>

- PR #2745³³³⁸ - Adding extended arithmetic performance counters
- PR #2744³³³⁹ - Adding ability to statistics counters to reset base counter
- Issue #2743³³⁴⁰ - Statistics counter does not support resetting
- PR #2742³³⁴¹ - Making sure Vc V2 builds without additional HPX configuration flags
- PR #2741³³⁴² - Deprecate unwrapped and implement unwrap and unwrapping
- PR #2740³³⁴³ - Coroutine stackoverflow detection for linux posix; Issue #2408
- PR #2739³³⁴⁴ - Add files via upload
- PR #2738³³⁴⁵ - Appveyor support
- PR #2737³³⁴⁶ - Fixing 2735
- Issue #2736³³⁴⁷ - 1d_hydro example doesn't work
- Issue #2735³³⁴⁸ - partitioned_vector_subview test failing
- PR #2734³³⁴⁹ - Add C++11 range utilities
- PR #2733³³⁵⁰ - Adapting iterator requirements for parallel algorithms
- PR #2732³³⁵¹ - Integrate C++ Co-arrays
- PR #2731³³⁵² - Adding on_migrated event handler to migratable component instances
- Issue #2729³³⁵³ - Add on_migrated() event handler to migratable components
- Issue #2728³³⁵⁴ - Why Projection is needed in parallel algorithms?
- PR #2727³³⁵⁵ - Cmake files for StackOverflow Detection
- PR #2726³³⁵⁶ - CMake for Stack Overflow Detection
- PR #2725³³⁵⁷ - Implemented segmented algorithms for partitioned vector
- PR #2724³³⁵⁸ - Fix examples in Action documentation
- PR #2723³³⁵⁹ - Enable lcos::channel<T>::register_as
- Issue #2722³³⁶⁰ - channel register_as() failing on compilation

³³³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2745>

³³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2744>

³³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2743>

³³⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2742>

³³⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2741>

³³⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2740>

³³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2739>

³³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2738>

³³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2737>

³³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2736>

³³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2735>

³³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2734>

³³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2733>

³³⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2732>

³³⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2731>

³³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/2729>

³³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2728>

³³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2727>

³³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2726>

³³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2725>

³³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2724>

³³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2723>

³³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2722>

- PR #2721³³⁶¹ - Mind map
- PR #2720³³⁶² - reorder forward declarations to get rid of C++14-only auto return types
- PR #2719³³⁶³ - Add documentation for partitioned_vector and add features in pack.hpp
- Issue #2718³³⁶⁴ - Some forward declarations in execution_fwd.hpp aren't C++11-compatible
- PR #2717³³⁶⁵ - Config support for fallthrough attribute
- PR #2716³³⁶⁶ - Implement parallel::partition_copy
- PR #2715³³⁶⁷ - initial import of icu string serializer
- PR #2714³³⁶⁸ - initial import of valarray serializer
- PR #2713³³⁶⁹ - Remove slashes before CMAKE_FILES_DIRECTORY variables
- PR #2712³³⁷⁰ - Fixing wait for 1751
- PR #2711³³⁷¹ - Adjust code for minimal supported GCC having been bumped to 4.9
- PR #2710³³⁷² - Adding code of conduct
- PR #2709³³⁷³ - Fixing UB in destroy tests
- PR #2708³³⁷⁴ - Add inline to prevent multiple definition issue
- Issue #2707³³⁷⁵ - Multiple defined symbols for task_block.hpp in VS2015
- PR #2706³³⁷⁶ - Adding .clang-format file
- PR #2704³³⁷⁷ - Add a synchronous mapping API
- Issue #2703³³⁷⁸ - Request: Add the .clang-format file to the repository
- Issue #2702³³⁷⁹ - STELLAR-GROUP/Vc slower than VCv1 possibly due to wrong instructions generated
- Issue #2701³³⁸⁰ - Datapar with STELLAR-GROUP/Vc requires obscure flag
- Issue #2700³³⁸¹ - Naming inconsistency in parallel algorithms
- Issue #2699³³⁸² - Iterator requirements are different from standard in parallel copy_if.
- PR #2698³³⁸³ - Properly releasing parcelport write handlers

³³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2721>

³³⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2720>

³³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2719>

³³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2718>

³³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2717>

³³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2716>

³³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2715>

³³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2714>

³³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2713>

³³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2712>

³³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2711>

³³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2710>

³³⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2709>

³³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2708>

³³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2707>

³³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2706>

³³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2704>

³³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2703>

³³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2702>

³³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2701>

³³⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/2700>

³³⁸² <https://github.com/STELLAR-GROUP/hpx/issues/2699>

³³⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2698>

- Issue #2697³³⁸⁴ - Compile error in addressing_service.cpp
- Issue #2696³³⁸⁵ - Building and using HPX statically: undefined references from runtime_support_server.cpp
- Issue #2695³³⁸⁶ - Executor changes cause compilation failures
- PR #2694³³⁸⁷ - Refining C++ language mode detection for MSVC
- PR #2693³³⁸⁸ - P0443 r2
- PR #2692³³⁸⁹ - Partially reverting changes to parcel_await
- Issue #2689³³⁹⁰ - HPX build fails when HPX_WITH_CUDA is enabled
- PR #2688³³⁹¹ - Make Cuda Clang builds pass
- PR #2687³³⁹² - Add an is_tuple_like trait for sequenceable type detection
- PR #2686³³⁹³ - Allowing throttling scheduler to be used without idle backoff
- PR #2685³³⁹⁴ - Add support of std::array to hpx::util::tuple_size and tuple_element
- PR #2684³³⁹⁵ - Adding new statistics performance counters
- PR #2683³³⁹⁶ - Replace boost::exception_ptr with std::exception_ptr
- Issue #2682³³⁹⁷ - HPX does not compile with HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF
- PR #2681³³⁹⁸ - Attempt to fix problem in managed_component_base
- PR #2680³³⁹⁹ - Fix bad size during archive creation
- Issue #2679³⁴⁰⁰ - Mismatch between size of archive and container
- Issue #2678³⁴⁰¹ - In parallel algorithm, other tasks are executed to the end even if an exception occurs in any task.
- PR #2677³⁴⁰² - Adding include check for std::addressof
- PR #2676³⁴⁰³ - Adding parallel::destroy and destroy_n
- PR #2675³⁴⁰⁴ - Making sure statistics counters work as expected
- PR #2674³⁴⁰⁵ - Turning assertions into exceptions
- PR #2673³⁴⁰⁶ - Inhibit direct conversion from future<future<T>> --> future<void>

³³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2697>

³³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2696>

³³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2695>

³³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2694>

³³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2693>

³³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2692>

³³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2689>

³³⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2688>

³³⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2687>

³³⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2686>

³³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2685>

³³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2684>

³³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2683>

³³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2682>

³³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2681>

³³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2680>

³⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2679>

³⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/2678>

³⁴⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2677>

³⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2676>

³⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2675>

³⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2674>

³⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2673>

- PR #2672³⁴⁰⁷ - C++17 invoke forms
- PR #2671³⁴⁰⁸ - Adding uninitialized_value_construct and uninitialized_value_construct_n
- PR #2670³⁴⁰⁹ - Integrate spmd multidimensional views for partitioned_vectors
- PR #2669³⁴¹⁰ - Adding uninitialized_default_construct and uninitialized_default_construct_n
- PR #2668³⁴¹¹ - Fixing documentation index
- Issue #2667³⁴¹² - Ambiguity of nested hpx::future<void>'s
- Issue #2666³⁴¹³ - Statistics Performance counter is not working
- PR #2664³⁴¹⁴ - Adding uninitialized_move and uninitialized_move_n
- Issue #2663³⁴¹⁵ - Seg fault in managed_component::get_base_gid, possibly cause by util::reinitializable_static
- Issue #2662³⁴¹⁶ - Crash in managed_component::get_base_gid due to problem with util::reinitializable_static
- PR #2665³⁴¹⁷ - Hide the detail namespace in doxygen per default
- PR #2660³⁴¹⁸ - Add documentation to hpx::util::unwrapped and hpx::util::unwrapped2
- PR #2659³⁴¹⁹ - Improve integration with vcpkg
- PR #2658³⁴²⁰ - Unify access_data trait for use in both, serialization and de-serialization
- PR #2657³⁴²¹ - Removing hpx::lcos::queue<T>
- PR #2656³⁴²² - Reduce MAX_TERMINATED_THREADS default, improve memory use on manycore cpus
- PR #2655³⁴²³ - Maintenaince for emulate-deleted macros
- PR #2654³⁴²⁴ - Implement parallel is_heap and is_heap_until
- PR #2653³⁴²⁵ - Drop support for VS2013
- PR #2652³⁴²⁶ - This patch makes sure that all parcels in a batch are properly handled
- PR #2649³⁴²⁷ - Update docs (Table 18) - move transform to end
- Issue #2647³⁴²⁸ - hpx::parcelset::detail::parcel_data::**has_continuation** is uninitialized
- Issue #2644³⁴²⁹ - Some .vcxproj in the HPX.sln fail to build

³⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2672>

³⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2671>

³⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2670>

³⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2669>

³⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2668>

³⁴¹² <https://github.com/STELLAR-GROUP/hpx/issues/2667>

³⁴¹³ <https://github.com/STELLAR-GROUP/hpx/issues/2666>

³⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2664>

³⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2663>

³⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2662>

³⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2665>

³⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2660>

³⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2659>

³⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2658>

³⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2657>

³⁴²² <https://github.com/STELLAR-GROUP/hpx/pull/2656>

³⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/2655>

³⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2654>

³⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2653>

³⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2652>

³⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2649>

³⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2647>

³⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2644>

- Issue #2641³⁴³⁰ - `hpx::lcos::queue` should be deprecated
- PR #2640³⁴³¹ - A new throttling policy with public APIs to suspend/resume
- PR #2639³⁴³² - Fix a tiny typo in tutorial.
- Issue #2638³⁴³³ - Invalid return type ‘void’ of constexpr function
- PR #2636³⁴³⁴ - Add and use HPX_MSVC_WARNING_PRAGMA for #pragma warning
- PR #2633³⁴³⁵ - Distributed define_spmd_block
- PR #2632³⁴³⁶ - Making sure container serialization uses size-compatible types
- PR #2631³⁴³⁷ - Add lcos::local::one_element_channel
- PR #2629³⁴³⁸ - Move unordered_map out of parcelport into hpx/concurrent
- PR #2628³⁴³⁹ - Making sure that shutdown does not hang
- PR #2627³⁴⁴⁰ - Fix serialization
- PR #2626³⁴⁴¹ - Generate `cmake_variables.qbk` and `cmake_toolchains.qbk` outside of the source tree
- PR #2625³⁴⁴² - Supporting `-std=c++17` flag
- PR #2624³⁴⁴³ - Fixing a small cmake typo
- PR #2622³⁴⁴⁴ - Update CMake minimum required version to 3.0.2 (closes #2621)
- Issue #2621³⁴⁴⁵ - Compiling hpx master fails with /usr/bin/ld: final link failed: Bad value
- PR #2620³⁴⁴⁶ - Remove warnings due to some captured variables
- PR #2619³⁴⁴⁷ - LF multiple parcels
- PR #2618³⁴⁴⁸ - Some fixes to libfabric that didn’t get caught before the merge
- PR #2617³⁴⁴⁹ - Adding `hpx::local_new`
- PR #2616³⁴⁵⁰ - Documentation: Extract all entities in order to autolink functions correctly
- Issue #2615³⁴⁵¹ - Documentation: Linking functions is broken
- PR #2614³⁴⁵² - Adding serialization for `std::deque`

³⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2641>

³⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2640>

³⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/2639>

³⁴³³ <https://github.com/STELLAR-GROUP/hpx/issues/2638>

³⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2636>

³⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2633>

³⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2632>

³⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2631>

³⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2629>

³⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2628>

³⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2627>

³⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2626>

³⁴⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2625>

³⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2624>

³⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2622>

³⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2621>

³⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2620>

³⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2619>

³⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2618>

³⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2617>

³⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2616>

³⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2615>

³⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2614>

- PR #2613³⁴⁵³ - We need to link with boost.thread and boost.chrono if we use boost.context
- PR #2612³⁴⁵⁴ - Making sure for_loop_n(par, ...) is actually executed in parallel
- PR #2611³⁴⁵⁵ - Add documentation to invoke_fused and friends NFC
- PR #2610³⁴⁵⁶ - Added reduction templates using an identity value
- PR #2608³⁴⁵⁷ - Fixing some unused vars in inspect
- PR #2607³⁴⁵⁸ - Fixed build for mingw
- PR #2606³⁴⁵⁹ - Supporting generic context for boost >= 1.61
- PR #2605³⁴⁶⁰ - Parcelport libfabric3
- PR #2604³⁴⁶¹ - Adding allocator support to promise and friends
- PR #2603³⁴⁶² - Barrier hang
- PR #2602³⁴⁶³ - Changes to scheduler to steal from one high-priority queue
- Issue #2601³⁴⁶⁴ - High priority tasks are not executed first
- PR #2600³⁴⁶⁵ - Compat fixes
- PR #2599³⁴⁶⁶ - Compatibility layer for threading support
- PR #2598³⁴⁶⁷ - V1.1
- PR #2597³⁴⁶⁸ - Release V1.0
- PR #2592³⁴⁶⁹ - First attempt to introduce spmd_block in hpx
- PR #2586³⁴⁷⁰ - local_segment in segmented_iterator_traits
- Issue #2584³⁴⁷¹ - Add allocator support to promise, packaged_task and friends
- PR #2576³⁴⁷² - Add missing dependencies of cuda based tests
- PR #2575³⁴⁷³ - Remove warnings due to some captured variables
- Issue #2574³⁴⁷⁴ - MSVC 2015 Compiler crash when building HPX
- Issue #2568³⁴⁷⁵ - Remove throttle_scheduler as it has been abandoned

³⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2613>

³⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2612>

³⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2611>

³⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2610>

³⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2608>

³⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2607>

³⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2606>

³⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2605>

³⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2604>

³⁴⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2603>

³⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2602>

³⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2601>

³⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2600>

³⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2599>

³⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2598>

³⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2597>

³⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2592>

³⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2586>

³⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/2584>

³⁴⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2576>

³⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2575>

³⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2574>

³⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2568>

- Issue #2566³⁴⁷⁶ - Add an inline versioning namespace before 1.0 release
- Issue #2565³⁴⁷⁷ - Raise minimal cmake version requirement
- PR #2556³⁴⁷⁸ - Fixing scan partitioner
- PR #2546³⁴⁷⁹ - Broadcast async
- Issue #2543³⁴⁸⁰ - make install fails due to a non-existing .so file
- PR #2495³⁴⁸¹ - wait_or_add_new returning thread_id_type
- Issue #2480³⁴⁸² - Unable to register new performance counter
- Issue #2471³⁴⁸³ - no type named ‘fcontext_t’ in namespace
- Issue #2456³⁴⁸⁴ - Re-implement hpx::util::unwrapped
- Issue #2455³⁴⁸⁵ - Add more arithmetic performance counters
- PR #2454³⁴⁸⁶ - Fix a couple of warnings and compiler errors
- PR #2453³⁴⁸⁷ - Timed executor support
- PR #2447³⁴⁸⁸ - Implementing new executor API (P0443)
- Issue #2439³⁴⁸⁹ - Implement executor proposal
- Issue #2408³⁴⁹⁰ - Stackoverflow detection for linux, e.g. based on libsigsegv
- PR #2377³⁴⁹¹ - Add a customization point for put_parcel so we can override actions
- Issue #2368³⁴⁹² - HPX_ASSERT problem
- Issue #2324³⁴⁹³ - Change default number of threads used to the maximum of the system
- Issue #2266³⁴⁹⁴ - hpx_0.9.99 make tests fail
- PR #2195³⁴⁹⁵ - Support for code completion in VIM
- Issue #2137³⁴⁹⁶ - Hpx does not compile over osx
- Issue #2092³⁴⁹⁷ - make tests should just build the tests
- Issue #2026³⁴⁹⁸ - Build HPX with Apple’s clang

³⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2566>

³⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2565>

³⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2556>

³⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2546>

³⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2543>

³⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2495>

³⁴⁸² <https://github.com/STELLAR-GROUP/hpx/issues/2480>

³⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/2471>

³⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2456>

³⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2455>

³⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2454>

³⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2453>

³⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2447>

³⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2439>

³⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2408>

³⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2377>

³⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/2368>

³⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/2324>

³⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2266>

³⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2195>

³⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2137>

³⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2092>

³⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2026>

- Issue #1932³⁴⁹⁹ - hpx with PBS fails on multiple localities
- PR #1914³⁵⁰⁰ - Parallel heap algorithm implementations WIP
- Issue #1598³⁵⁰¹ - Disconnecting a locality results in segfault using heartbeat example
- Issue #1404³⁵⁰² - unwrapped doesn't work with movable only types
- Issue #1400³⁵⁰³ - hpx::util::unwrapped doesn't work with non-future types
- Issue #1205³⁵⁰⁴ - TSS is broken
- Issue #1126³⁵⁰⁵ - vector<future<T>> does not work gracefully with dataflow, when_all and unwrapped
- Issue #1056³⁵⁰⁶ - Thread manager cleanup
- Issue #863³⁵⁰⁷ - Futures should not require a default constructor
- Issue #856³⁵⁰⁸ - Allow runtimemode_connect to be used with security enabled
- Issue #726³⁵⁰⁹ - Valgrind
- Issue #701³⁵¹⁰ - Add RCR performance counter component
- Issue #528³⁵¹¹ - Add support for known failures and warning count/comparisons to hpx_run_tests.py

2.10.15 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 #2289³⁵¹² 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.

³⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1932>

³⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1914>

³⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1598>

³⁵⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1404>

³⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1400>

³⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1205>

3505 <https://github.com/STELLAR-GROUP/hpx/issues/1126>3506 <https://github.com/STELLAR-GROUP/hpx/issues/1056>3507 <https://github.com/STELLAR-GROUP/hpx/issues/863>3508 <https://github.com/STELLAR-GROUP/hpx/issues/856>3509 <https://github.com/STELLAR-GROUP/hpx/issues/726>3510 <https://github.com/STELLAR-GROUP/hpx/issues/701>3511 <https://github.com/STELLAR-GROUP/hpx/issues/528>3512 <https://github.com/STELLAR-GROUP/hpx/pull/2289>

- We have started to add support for vectorization to our parallel algorithm implementations. This support depends on using an external library, currently either Vc Library or [boost_simd](#). Please see Issue #2333³⁵¹³ 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-post³⁵¹⁴ 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.
- 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.

³⁵¹³ <https://github.com/STELLAR-GROUP/hpx/issues/2333>

³⁵¹⁴ <http://stellar-group.org/2016/09/vectorized-cpp-parallel-algorithms-with-hpx/>

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2596³⁵¹⁵ - Adding apex data
- PR #2595³⁵¹⁶ - Remove obsolete file
- Issue #2594³⁵¹⁷ - FindOpenCL.cmake mismatch with the official cmake module
- PR #2592³⁵¹⁸ - First attempt to introduce spmd_block in hpx
- Issue #2591³⁵¹⁹ - Feature request: continuation (then) which does not require the callable object to take a future<R> as parameter
- PR #2588³⁵²⁰ - Daint fixes
- PR #2587³⁵²¹ - Fixing transfer_(continuation)_action::schedule
- PR #2585³⁵²² - Work around MSVC having an ICE when compiling with -Ob2
- PR #2583³⁵²³ - changing 7zip command to 7za in roll_release.sh
- PR #2582³⁵²⁴ - First attempt to introduce spmd_block in hpx
- PR #2581³⁵²⁵ - Enable annotated function for parallel algorithms
- PR #2580³⁵²⁶ - First attempt to introduce spmd_block in hpx
- PR #2579³⁵²⁷ - Make thread NICE level setting an option
- PR #2578³⁵²⁸ - Implementing enqueue instead of busy wait when no sender is available
- PR #2577³⁵²⁹ - Retrieve -std=c++11 consistent nvcc flag
- PR #2576³⁵³⁰ - Add missing dependencies of cuda based tests
- PR #2575³⁵³¹ - Remove warnings due to some captured variables
- PR #2573³⁵³² - Attempt to resolve resolve_locality
- PR #2572³⁵³³ - Adding APEX hooks to background thread
- PR #2571³⁵³⁴ - Pick up hpx.ignore_batch_env from config map
- PR #2570³⁵³⁵ - Add commandline options --hpx:print-counters-locally

³⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2596>

³⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2595>

³⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2594>

³⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2592>

³⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2591>

³⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2588>

³⁵²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2587>

³⁵²² <https://github.com/STELLAR-GROUP/hpx/pull/2585>

³⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/2583>

³⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2582>

³⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2581>

³⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2580>

³⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2579>

³⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2578>

³⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2577>

³⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2576>

³⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2575>

³⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/2573>

³⁵³³ <https://github.com/STELLAR-GROUP/hpx/pull/2572>

³⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2571>

³⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2570>

- PR #2569³⁵³⁶ - Fix computeapi unit tests
- PR #2567³⁵³⁷ - This adds another barrier::synchronize before registering performance counters
- PR #2564³⁵³⁸ - Cray static toolchain support
- PR #2563³⁵³⁹ - Fixed unhandled exception during startup
- PR #2562³⁵⁴⁰ - Remove partitioned_vector.cu from build tree when nvcc is used
- Issue #2561³⁵⁴¹ - octo-tiger crash with commit 6e921495ff6c26f125d62629cbaad0525f14f7ab
- PR #2560³⁵⁴² - Prevent -Wundef warnings on Vc version checks
- PR #2559³⁵⁴³ - Allowing CUDA callback to set the future directly from an OS thread
- PR #2558³⁵⁴⁴ - Remove warnings due to float precisions
- PR #2557³⁵⁴⁵ - Removing bogus handling of compile flags for CUDA
- PR #2556³⁵⁴⁶ - Fixing scan partitioner
- PR #2554³⁵⁴⁷ - Add more diagnostics to error thrown from find_appropriate_destination
- Issue #2555³⁵⁴⁸ - No valid parcelport configured
- PR #2553³⁵⁴⁹ - Add cmake cuda_arch option
- PR #2552³⁵⁵⁰ - Remove incomplete datapar bindings to libflatarray
- PR #2551³⁵⁵¹ - Rename hwloc_topology to hwloc_topology_info
- PR #2550³⁵⁵² - Apex api updates
- PR #2549³⁵⁵³ - Pre-include defines.hpp to get the macro HPX_HAVE_CUDA value
- PR #2548³⁵⁵⁴ - Fixing issue with disconnect
- PR #2546³⁵⁵⁵ - Some fixes around cuda clang partitioned_vector example
- PR #2545³⁵⁵⁶ - Fix uses of the Vc2 datapar flags; the value, not the type, should be passed to functions
- PR #2542³⁵⁵⁷ - Make HPX_WITH_MALLOC easier to use
- PR #2541³⁵⁵⁸ - avoid recompiles when enabling/disabling examples

³⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2569>

³⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2567>

³⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2564>

³⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2563>

³⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2562>

³⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/2561>

³⁵⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2560>

³⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2559>

³⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2558>

³⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2557>

³⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2556>

³⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2554>

³⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2555>

³⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2553>

³⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2552>

³⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2551>

³⁵⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2550>

³⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2549>

³⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2548>

³⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2546>

³⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2545>

³⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2542>

³⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2541>

- PR #2540³⁵⁵⁹ - Fixing usage of target_link_libraries()
- PR #2539³⁵⁶⁰ - fix RPATH behaviour
- Issue #2538³⁵⁶¹ - HPX_WITH_CUDA corrupts compilation flags
- PR #2537³⁵⁶² - Add output of a Bazel Skylark extension for paths and compile options
- PR #2536³⁵⁶³ - Add counter exposing total available memory to Windows as well
- PR #2535³⁵⁶⁴ - Remove obsolete support for security
- Issue #2534³⁵⁶⁵ - Remove command line option --hpx:run-agas-server
- PR #2533³⁵⁶⁶ - Pre-cache locality endpoints during bootstrap
- PR #2532³⁵⁶⁷ - Fixing handling of GIDs during serialization preprocessing
- PR #2531³⁵⁶⁸ - Amend uses of the term “functor”
- PR #2529³⁵⁶⁹ - added counter for reading available memory
- PR #2527³⁵⁷⁰ - Facilities to create actions from lambdas
- PR #2526³⁵⁷¹ - Updated docs: HPX_WITH_EXAMPLES
- PR #2525³⁵⁷² - Remove warnings related to unused captured variables
- Issue #2524³⁵⁷³ - CMAKE failed because it is missing: TCMALLOC_LIBRARY TCMALLOC_INCLUDE_DIR
- PR #2523³⁵⁷⁴ - Fixing compose_cb stack overflow
- PR #2522³⁵⁷⁵ - Instead of unlocking, ignore the lock while creating the message handler
- PR #2521³⁵⁷⁶ - Create LPROGRESS_ logging macro to simplify progress tracking and timings
- PR #2520³⁵⁷⁷ - Intel 17 support
- PR #2519³⁵⁷⁸ - Fix components example
- PR #2518³⁵⁷⁹ - Fixing parcel scheduling
- Issue #2517³⁵⁸⁰ - Race condition during Parcel Coalescing Handler creation
- Issue #2516³⁵⁸¹ - HPX locks up when using at least 256 localities

³⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2540>

³⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2539>

³⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2538>

³⁵⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2537>

³⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2536>

³⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2535>

³⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2534>

³⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2533>

³⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2532>

3568 <https://github.com/STELLAR-GROUP/hpx/pull/2531>3569 <https://github.com/STELLAR-GROUP/hpx/pull/2529>3570 <https://github.com/STELLAR-GROUP/hpx/pull/2527>3571 <https://github.com/STELLAR-GROUP/hpx/pull/2526>3572 <https://github.com/STELLAR-GROUP/hpx/pull/2525>3573 <https://github.com/STELLAR-GROUP/hpx/issues/2524>3574 <https://github.com/STELLAR-GROUP/hpx/pull/2523>3575 <https://github.com/STELLAR-GROUP/hpx/pull/2522>3576 <https://github.com/STELLAR-GROUP/hpx/pull/2521>3577 <https://github.com/STELLAR-GROUP/hpx/pull/2520>3578 <https://github.com/STELLAR-GROUP/hpx/pull/2519>3579 <https://github.com/STELLAR-GROUP/hpx/pull/2518>3580 <https://github.com/STELLAR-GROUP/hpx/issues/2517>3581 <https://github.com/STELLAR-GROUP/hpx/issues/2516>

- Issue #2515³⁵⁸² - error: Install cannot find “/lib/hpx/libparcel_coalescing.so.0.9.99” but I can see that file
- PR #2514³⁵⁸³ - Making sure that all continuations of a shared_future are invoked in order
- PR #2513³⁵⁸⁴ - Fixing locks held during suspension
- PR #2512³⁵⁸⁵ - MPI Parcelport improvements and fixes related to the background work changes
- PR #2511³⁵⁸⁶ - Fixing bit-wise (zero-copy) serialization
- Issue #2509³⁵⁸⁷ - Linking errors in hwloc_topology
- PR #2508³⁵⁸⁸ - Added documentation for debugging with core files
- PR #2506³⁵⁸⁹ - Fixing background work invocations
- PR #2505³⁵⁹⁰ - Fix tuple serialization
- Issue #2504³⁵⁹¹ - Ensure continuations are called in the order they have been attached
- PR #2503³⁵⁹² - Adding serialization support for Vc v2 (datapar)
- PR #2502³⁵⁹³ - Resolve various, minor compiler warnings
- PR #2501³⁵⁹⁴ - Some other fixes around cuda examples
- Issue #2500³⁵⁹⁵ - nvcc / cuda clang issue due to a missing -DHPX_WITH_CUDA flag
- PR #2499³⁵⁹⁶ - Adding support for std::array to wait_all and friends
- PR #2498³⁵⁹⁷ - Execute background work as HPX thread
- PR #2497³⁵⁹⁸ - Fixing configuration options for spinlock-deadlock detection
- PR #2496³⁵⁹⁹ - Accounting for different compilers in CrayKNL toolchain file
- PR #2494³⁶⁰⁰ - Adding component base class which ties a component instance to a given executor
- PR #2493³⁶⁰¹ - Enable controlling amount of pending threads which must be available to allow thread stealing
- PR #2492³⁶⁰² - Adding new command line option –hpx:print-counter-reset
- PR #2491³⁶⁰³ - Resolve ambiguities when compiling with APEX
- PR #2490³⁶⁰⁴ - Resuming threads waiting on future with higher priority

³⁵⁸² <https://github.com/STELLAR-GROUP/hpx/issues/2515>

³⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2514>

³⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2513>

³⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2512>

³⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2511>

³⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2509>

³⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2508>

³⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2506>

³⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2505>

³⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2504>

³⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2503>

³⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2502>

³⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2501>

³⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2500>

³⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2499>

³⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2498>

³⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2497>

³⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2496>

³⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2494>

³⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2493>

³⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2492>

³⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2491>

³⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2490>

- Issue #2489³⁶⁰⁵ - nvcc issue because -std=c++11 appears twice
- PR #2488³⁶⁰⁶ - Adding performance counters exposing the internal idle and busy-loop counters
- PR #2487³⁶⁰⁷ - Allowing for plain suspend to reschedule thread right away
- PR #2486³⁶⁰⁸ - Only flag HPX code for CUDA if HPX_WITH_CUDA is set
- PR #2485³⁶⁰⁹ - Making thread-queue parameters runtime-configurable
- PR #2484³⁶¹⁰ - Added atomic counter for parcel-destinations
- PR #2483³⁶¹¹ - Added priority-queue lifo scheduler
- PR #2482³⁶¹² - Changing scheduler to steal only if more than a minimal number of tasks are available
- PR #2481³⁶¹³ - Extending command line option –hpx:print-counter-destination to support value ‘none’
- PR #2479³⁶¹⁴ - Added option to disable signal handler
- PR #2478³⁶¹⁵ - Making sure the sine performance counter module gets loaded only for the corresponding example
- Issue #2477³⁶¹⁶ - Breaking at a throw statement
- PR #2476³⁶¹⁷ - Annotated function
- PR #2475³⁶¹⁸ - Ensure that using %osthread% during logging will not throw for non-hpx threads
- PR #2474³⁶¹⁹ - Remove now superficial non_direct actions from base_lco and friends
- PR #2473³⁶²⁰ - Refining support for ITTNotify
- PR #2472³⁶²¹ - Some fixes around hpx compute
- Issue #2470³⁶²² - redefinition of boost::detail::spinlock
- Issue #2469³⁶²³ - Dataflow performance issue
- PR #2468³⁶²⁴ - Perf docs update
- PR #2466³⁶²⁵ - Guarantee to execute remote direct actions on HPX-thread
- PR #2465³⁶²⁶ - Improve demo : Async copy and fixed device handling
- PR #2464³⁶²⁷ - Adding performance counter exposing instantaneous scheduler utilization

³⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2489>

³⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2488>

³⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2487>

³⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2486>

³⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2485>

³⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2484>

³⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2483>

³⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/2482>

³⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2481>

³⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2479>

³⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2478>

³⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2477>

³⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2476>

³⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2475>

³⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2474>

³⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2473>

³⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2472>

³⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/2470>

³⁶²³ <https://github.com/STELLAR-GROUP/hpx/issues/2469>

³⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2468>

³⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2466>

³⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2465>

³⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2464>

- PR #2463³⁶²⁸ - Downcast to future<void>
- PR #2462³⁶²⁹ - Fixed usage of ITT-Notify API with Intel Amplifier
- PR #2461³⁶³⁰ - Cublas demo
- PR #2460³⁶³¹ - Fixing thread bindings
- PR #2459³⁶³² - Make -std=c++11 nvcc flag consistent for in-build and installed versions
- Issue #2457³⁶³³ - Segmentation fault when registering a partitioned vector
- PR #2452³⁶³⁴ - Properly releasing global barrier for unhandled exceptions
- PR #2451³⁶³⁵ - Fixing long shutdown times
- PR #2450³⁶³⁶ - Attempting to fix initialization errors on newer platforms (Boost V1.63)
- PR #2449³⁶³⁷ - Replace BOOST_COMPILER_FENCE with an HPX version
- PR #2448³⁶³⁸ - This fixes a possible race in the migration code
- PR #2445³⁶³⁹ - **Fixing dataflow et.al. for futures or future-ranges wrapped into ref()**
- PR #2444³⁶⁴⁰ - Fix segfaults
- PR #2443³⁶⁴¹ - Issue 2442
- Issue #2442³⁶⁴² - Mismatch between #if/#endif and namespace scope brackets in this_thread_executers.hpp
- Issue #2441³⁶⁴³ - undeclared identifier BOOST_COMPILER_FENCE
- PR #2440³⁶⁴⁴ - Knl build
- PR #2438³⁶⁴⁵ - Datapar backend
- PR #2437³⁶⁴⁶ - Adapt algorithm parameter sequence changes from C++17
- PR #2436³⁶⁴⁷ - Adapt execution policy name changes from C++17
- Issue #2435³⁶⁴⁸ - Trunk broken, undefined reference to hpx::thread::interrupt(hpx::thread::id, bool)
- PR #2434³⁶⁴⁹ - More fixes to resource manager
- PR #2433³⁶⁵⁰ - Added versions of hpx::get_ptr taking client side representations

³⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2463>

³⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2462>

³⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2461>

³⁶³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2460>

³⁶³² <https://github.com/STELLAR-GROUP/hpx/pull/2459>

³⁶³³ <https://github.com/STELLAR-GROUP/hpx/issues/2457>

³⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2452>

³⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2451>

³⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2450>

³⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2449>

³⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2448>

³⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2445>

³⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2444>

³⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2443>

³⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/2442>

³⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/2441>

³⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2440>

³⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2438>

³⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2437>

³⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2436>

³⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2435>

³⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2434>

³⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2433>

- PR #2432³⁶⁵¹ - Warning fixes
- PR #2431³⁶⁵² - Adding facility representing set of performance counters
- PR #2430³⁶⁵³ - Fix parallel_executor thread spawning
- PR #2429³⁶⁵⁴ - Fix attribute warning for gcc
- Issue #2427³⁶⁵⁵ - Seg fault running octo-tiger with latest HPX commit
- Issue #2426³⁶⁵⁶ - Bug in 9592f5c0bc29806fce0dbe73f35b6ca7e027edcb causes immediate crash in Octo-tiger
- PR #2425³⁶⁵⁷ - Fix nvcc errors due to constexpr specifier
- Issue #2424³⁶⁵⁸ - Async action on component present on hpx::find_here is executing synchronously
- PR #2423³⁶⁵⁹ - Fix nvcc errors due to constexpr specifier
- PR #2422³⁶⁶⁰ - Implementing hpx::this_thread thread data functions
- PR #2421³⁶⁶¹ - Adding benchmark for wait_all
- Issue #2420³⁶⁶² - Returning object of a component client from another component action fails
- PR #2419³⁶⁶³ - Infiniband parcelport
- Issue #2418³⁶⁶⁴ - gcc + nvcc fails to compile code that uses partitioned_vector
- PR #2417³⁶⁶⁵ - Fixing context switching
- PR #2416³⁶⁶⁶ - Adding fixes and workarounds to allow compilation with nvcc/msvc (VS2015up3)
- PR #2415³⁶⁶⁷ - Fix errors coming from hpx compute examples
- PR #2414³⁶⁶⁸ - Fixing msvc12
- PR #2413³⁶⁶⁹ - Enable cuda/nvcc or cuda/clang when using add_hpx_executable()
- PR #2412³⁶⁷⁰ - Fix issue in HPX_SetupTarget.cmake when cuda is used
- PR #2411³⁶⁷¹ - This fixes the core compilation issues with MSVC12
- Issue #2410³⁶⁷² - undefined reference to opal_hwloc191_hwloc_.....
- PR #2409³⁶⁷³ - Fixing locking for channel and receive_buffer

³⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2432>

³⁶⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2431>

³⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2430>

³⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2429>

³⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2427>

³⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2426>

³⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2425>

³⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2424>

³⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2423>

³⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2422>

³⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2421>

³⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2420>

³⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2419>

³⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2418>

³⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2417>

³⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2416>

³⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2415>

³⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2414>

³⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2413>

³⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2412>

³⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2411>

³⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/2410>

³⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2409>

- PR #2407³⁶⁷⁴ - Solving #2402 and #2403
- PR #2406³⁶⁷⁵ - Improve guards
- PR #2405³⁶⁷⁶ - Enable parallel::for_each for iterators returning proxy types
- PR #2404³⁶⁷⁷ - Forward the explicitly given result_type in the hpx invoke
- Issue #2403³⁶⁷⁸ - datapar_execution + zip iterator: lambda arguments aren't references
- Issue #2402³⁶⁷⁹ - datapar algorithm instantiated with wrong type #2402
- PR #2401³⁶⁸⁰ - Added support for imported libraries to HPX_Libraries.cmake
- PR #2400³⁶⁸¹ - Use CMake policy CMP0060
- Issue #2399³⁶⁸² - Error trying to push back vector of futures to vector
- PR #2398³⁶⁸³ - Allow config #defines to be written out to custom config/defines.hpp
- Issue #2397³⁶⁸⁴ - CMake generated config defines can cause tedious rebuilds category
- Issue #2396³⁶⁸⁵ - BOOST_ROOT paths are not used at link time
- PR #2395³⁶⁸⁶ - Fix target_link_libraries() issue when HPX Cuda is enabled
- Issue #2394³⁶⁸⁷ - Template compilation error using HPX_WITH_DATAPAR_LIBFLATARRAY
- PR #2393³⁶⁸⁸ - Fixing lock registration for recursive mutex
- PR #2392³⁶⁸⁹ - Add keywords in target_link_libraries in hpx_setup_target
- PR #2391³⁶⁹⁰ - Clang goroutines
- Issue #2390³⁶⁹¹ - Adapt execution policy name changes from C++17
- PR #2389³⁶⁹² - Chunk allocator and pool are not used and are obsolete
- PR #2388³⁶⁹³ - Adding functionalities to datapar needed by octotiger
- PR #2387³⁶⁹⁴ - Fixing race condition for early parcels
- Issue #2386³⁶⁹⁵ - Lock registration broken for recursive_mutex
- PR #2385³⁶⁹⁶ - Datapar zip iterator

³⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2407>

³⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2406>

³⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2405>

³⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2404>

³⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2403>

³⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2402>

³⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2401>

³⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2400>

³⁶⁸² <https://github.com/STELLAR-GROUP/hpx/issues/2399>

³⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2398>

³⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2397>

³⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2396>

³⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2395>

³⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2394>

³⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2393>

³⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2392>

³⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2391>

³⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2390>

³⁶⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2389>

³⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2388>

³⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2387>

³⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2386>

³⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2385>

- PR #2384³⁶⁹⁷ - Fixing race condition in for_loop_reduction
- PR #2383³⁶⁹⁸ - Continuations
- PR #2382³⁶⁹⁹ - add LibFlatArray-based backend for datapar
- PR #2381³⁷⁰⁰ - remove unused typedef to get rid of compiler warnings
- PR #2380³⁷⁰¹ - Tau cleanup
- PR #2379³⁷⁰² - Can send immediate
- PR #2378³⁷⁰³ - Renaming copy_helper/copy_n_helper/move_helper/move_n_helper
- Issue #2376³⁷⁰⁴ - Boost trunk's spinlock initializer fails to compile
- PR #2375³⁷⁰⁵ - Add support for minimal thread local data
- PR #2374³⁷⁰⁶ - Adding API functions set_config_entry_callback
- PR #2373³⁷⁰⁷ - Add a simple utility for debugging that gives suspended task backtraces
- PR #2372³⁷⁰⁸ - Barrier Fixes
- Issue #2370³⁷⁰⁹ - Can't wait on a wrapped future
- PR #2369³⁷¹⁰ - Fixing stable_partition
- PR #2367³⁷¹¹ - Fixing find_prefixes for Windows platforms
- PR #2366³⁷¹² - Testing for experimental/optional only in C++14 mode
- PR #2364³⁷¹³ - Adding set_config_entry
- PR #2363³⁷¹⁴ - Fix papi
- PR #2362³⁷¹⁵ - Adding missing macros for new non-direct actions
- PR #2361³⁷¹⁶ - Improve cmake output to help debug compiler incompatibility check
- PR #2360³⁷¹⁷ - Fixing race condition in condition_variable
- PR #2359³⁷¹⁸ - Fixing shutdown when parcels are still in flight
- Issue #2357³⁷¹⁹ - failed to insert console_print_action into typename_to_id_t registry

³⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2384>

³⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2383>

³⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2382>

³⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2381>

³⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2380>

³⁷⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2379>

³⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2378>

³⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2376>

³⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2375>

3706 <https://github.com/STELLAR-GROUP/hpx/pull/2374>3707 <https://github.com/STELLAR-GROUP/hpx/pull/2373>3708 <https://github.com/STELLAR-GROUP/hpx/pull/2372>3709 <https://github.com/STELLAR-GROUP/hpx/issues/2370>3710 <https://github.com/STELLAR-GROUP/hpx/pull/2369>3711 <https://github.com/STELLAR-GROUP/hpx/pull/2367>3712 <https://github.com/STELLAR-GROUP/hpx/pull/2366>3713 <https://github.com/STELLAR-GROUP/hpx/pull/2364>3714 <https://github.com/STELLAR-GROUP/hpx/pull/2363>3715 <https://github.com/STELLAR-GROUP/hpx/pull/2362>3716 <https://github.com/STELLAR-GROUP/hpx/pull/2361>3717 <https://github.com/STELLAR-GROUP/hpx/pull/2360>3718 <https://github.com/STELLAR-GROUP/hpx/pull/2359>3719 <https://github.com/STELLAR-GROUP/hpx/issues/2357>

- PR #2356³⁷²⁰ - Fixing return type of get_iterator_tuple
- PR #2355³⁷²¹ - Fixing compilation against Boost 1.62
- PR #2354³⁷²² - Adding serialization for mask_type if CPU_COUNT > 64
- PR #2353³⁷²³ - Adding hooks to tie in APEX into the parcel layer
- Issue #2352³⁷²⁴ - Compile errors when using intel 17 beta (for KNL) on edison
- PR #2351³⁷²⁵ - Fix function vtable get_function_address implementation
- Issue #2350³⁷²⁶ - Build failure - master branch (4de09f5) with Intel Compiler v17
- PR #2349³⁷²⁷ - Enabling zero-copy serialization support for std::vector<>
- PR #2348³⁷²⁸ - Adding test to verify #2334 is fixed
- PR #2347³⁷²⁹ - Bug fixes for hpx.compute and hpx::lcos::channel
- PR #2346³⁷³⁰ - Removing cmake “find” files that are in the APEX cmake Modules
- PR #2345³⁷³¹ - Implemented parallel::stable_partition
- PR #2344³⁷³² - Making hpx::lcos::channel usable with basename registration
- PR #2343³⁷³³ - Fix a couple of examples that failed to compile after recent api changes
- Issue #2342³⁷³⁴ - Enabling APEX causes link errors
- PR #2341³⁷³⁵ - Removing cmake “find” files that are in the APEX cmake Modules
- PR #2340³⁷³⁶ - Implemented all existing datapar algorithms using Boost.SIMD
- PR #2339³⁷³⁷ - Fixing 2338
- PR #2338³⁷³⁸ - Possible race in sliding semaphore
- PR #2337³⁷³⁹ - Adjust osu_latency test to measure window_size parcels in flight at once
- PR #2336³⁷⁴⁰ - Allowing remote direct actions to be executed without spawning a task
- PR #2335³⁷⁴¹ - Making sure multiple components are properly initialized from arguments
- Issue #2334³⁷⁴² - Cannot construct component with large vector on a remote locality

³⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2356>

³⁷²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2355>

³⁷²² <https://github.com/STELLAR-GROUP/hpx/pull/2354>

³⁷²³ <https://github.com/STELLAR-GROUP/hpx/pull/2353>

³⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2352>

³⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2351>

³⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2350>

³⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2349>

³⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2348>

³⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2347>

³⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2346>

³⁷³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2345>

³⁷³² <https://github.com/STELLAR-GROUP/hpx/pull/2344>

³⁷³³ <https://github.com/STELLAR-GROUP/hpx/pull/2343>

³⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2342>

³⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2341>

³⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2340>

³⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2339>

³⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2338>

³⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2337>

³⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2336>

³⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2335>

³⁷⁴² <https://github.com/STELLAR-GROUP/hpx/issues/2334>

- PR #2332³⁷⁴³ - Fixing hpx::lcos::local::barrier
- PR #2331³⁷⁴⁴ - Updating APEX support to include OTF2
- PR #2330³⁷⁴⁵ - Support for data-parallelism for parallel algorithms
- Issue #2329³⁷⁴⁶ - Coordinate settings in cmake
- PR #2328³⁷⁴⁷ - fix LibGeoDecomp builds with HPX + GCC 5.3.0 + CUDA 8RC
- PR #2326³⁷⁴⁸ - Making scan_partitioner work (for now)
- Issue #2323³⁷⁴⁹ - Constructing a vector of components only correctly initializes the first component
- PR #2322³⁷⁵⁰ - Fix problems that bubbled up after merging #2278
- PR #2321³⁷⁵¹ - Scalable barrier
- PR #2320³⁷⁵² - Std flag fixes
- Issue #2319³⁷⁵³ - -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 #2318³⁷⁵⁴ - Improve handling of -hpx:bind=<bind-spec>
- PR #2317³⁷⁵⁵ - Making sure command line warnings are printed once only
- PR #2316³⁷⁵⁶ - Fixing command line handling for default bind mode
- PR #2315³⁷⁵⁷ - Set id_retrieved if set_id is present
- Issue #2314³⁷⁵⁸ - Warning for requested/allocated thread discrepancy is printed twice
- Issue #2313³⁷⁵⁹ - -hpx:print-bind doesn't work with -hpx:pu-step
- Issue #2312³⁷⁶⁰ - -hpx:bind range specifier restrictions are overly restrictive
- Issue #2311³⁷⁶¹ - hpx_0.9.99 out of project build fails
- PR #2310³⁷⁶² - Simplify function registration
- PR #2309³⁷⁶³ - Spelling and grammar revisions in documentation (and some code)
- PR #2306³⁷⁶⁴ - Correct minor typo in the documentation
- PR #2305³⁷⁶⁵ - Cleaning up and fixing parcel coalescing

³⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2332>

³⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2331>

³⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2330>

³⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2329>

³⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2328>

³⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2326>

³⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2323>

³⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2322>

³⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2321>

³⁷⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2320>

³⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/2319>

³⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2318>

³⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2317>

³⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2316>

³⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2315>

³⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2314>

³⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2313>

³⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2312>

³⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2311>

³⁷⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2310>

³⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2309>

³⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2306>

³⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2305>

- PR #2304³⁷⁶⁶ - Inspect checks for stream related includes
- PR #2303³⁷⁶⁷ - Add functionality allowing to enumerate threads of given state
- PR #2301³⁷⁶⁸ - Algorithm overloads fix for VS2013
- PR #2300³⁷⁶⁹ - Use <cstdint>, add inspect checks
- PR #2299³⁷⁷⁰ - Replace boost::[c]ref with std:::[c]ref, add inspect checks
- PR #2297³⁷⁷¹ - Fixing compilation with no hw_loc
- PR #2296³⁷⁷² - Hpx compute
- PR #2295³⁷⁷³ - Making sure for_loop(execution::par, 0, N, ...) is actually executed in parallel
- PR #2294³⁷⁷⁴ - Throwing exceptions if the runtime is not up and running
- PR #2293³⁷⁷⁵ - Removing unused parcel port code
- PR #2292³⁷⁷⁶ - Refactor function vtables
- PR #2291³⁷⁷⁷ - Fixing 2286
- PR #2290³⁷⁷⁸ - Simplify algorithm overloads
- PR #2289³⁷⁷⁹ - Adding performance counters reporting parcel related data on a per-action basis
- Issue #2288³⁷⁸⁰ - Remove dormant parcelports
- Issue #2286³⁷⁸¹ - adjustments to parcel handling to support parcelports that do not need a connection cache
- PR #2285³⁷⁸² - add CMake option to disable package export
- PR #2283³⁷⁸³ - Add more inspect checks for use of deprecated components
- Issue #2282³⁷⁸⁴ - Arithmetic exception in executor static chunker
- Issue #2281³⁷⁸⁵ - For loop doesn't parallelize
- PR #2280³⁷⁸⁶ - Fixing 2277: build failure with PAPI
- PR #2279³⁷⁸⁷ - Child vs parent stealing
- Issue #2277³⁷⁸⁸ - master branch build failure (53c5b4f) with papi

³⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2304>

³⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2303>

³⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2301>

³⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2300>

³⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2299>

³⁷⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2297>

³⁷⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2296>

³⁷⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2295>

³⁷⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2294>

³⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2293>

³⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2292>

³⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2291>

³⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2290>

³⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2289>

³⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2288>

³⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/2286>

³⁷⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2285>

³⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2283>

³⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2282>

³⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2281>

³⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2280>

³⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2279>

³⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2277>

- PR #2276³⁷⁸⁹ - Compile time launch policies
- PR #2275³⁷⁹⁰ - Replace boost::chrono with std::chrono in interfaces
- PR #2274³⁷⁹¹ - Replace most uses of Boost.Assign with initializer list
- PR #2273³⁷⁹² - Fixed typos
- PR #2272³⁷⁹³ - Inspect checks
- PR #2270³⁷⁹⁴ - Adding test verifying -Ihpx.os_threads=all
- PR #2269³⁷⁹⁵ - Added inspect check for now obsolete boost type traits
- PR #2268³⁷⁹⁶ - Moving more code into source files
- Issue #2267³⁷⁹⁷ - Add inspect support to deprecate Boost.TypeTraits
- PR #2265³⁷⁹⁸ - Adding channel LCO
- PR #2264³⁷⁹⁹ - Make support for std::ref mandatory
- PR #2263³⁸⁰⁰ - Constrain tuple_member forwarding constructor
- Issue #2262³⁸⁰¹ - Test hpx.os_threads=all
- Issue #2261³⁸⁰² - OS X: Error: no matching constructor for initialization of 'hpx::lcos::local::condition_variable_any'
- Issue #2260³⁸⁰³ - Make support for std::ref mandatory
- PR #2259³⁸⁰⁴ - Remove most of Boost.MPL, Boost.EnableIf and Boost.TypeTraits
- PR #2258³⁸⁰⁵ - Fixing #2256
- PR #2257³⁸⁰⁶ - Fixing launch process
- Issue #2256³⁸⁰⁷ - Actions are not registered if not invoked
- PR #2255³⁸⁰⁸ - Coalescing histogram
- PR #2254³⁸⁰⁹ - Silence explicit initialization in copy-constructor warnings
- PR #2253³⁸¹⁰ - Drop support for GCC 4.6 and 4.7
- PR #2252³⁸¹¹ - Prepare V1.0

³⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2276>

³⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2275>

³⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2274>

³⁷⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2273>

³⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/2272>

³⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2270>

³⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2269>

³⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2268>

³⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2267>

³⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2265>

³⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2264>

³⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2263>

³⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/2262>

³⁸⁰² <https://github.com/STELLAR-GROUP/hpx/issues/2261>

³⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2260>

³⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2259>

³⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2258>

³⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2257>

³⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2256>

³⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2255>

³⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2254>

³⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2253>

³⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2252>

- PR #2251³⁸¹² - Convert to 0.9.99
- PR #2249³⁸¹³ - Adding iterator_facade and iterator_adaptor
- Issue #2248³⁸¹⁴ - Need a feature to yield to a new task immediately
- PR #2246³⁸¹⁵ - Adding split_future
- PR #2245³⁸¹⁶ - Add an example for handing over a component instance to a dynamically launched locality
- Issue #2243³⁸¹⁷ - Add example demonstrating AGAS symbolic name registration
- Issue #2242³⁸¹⁸ - pkgconfig test broken on CentOS 7 / Boost 1.61
- Issue #2241³⁸¹⁹ - Compilation error for partitioned vector in hpx_compute branch
- PR #2240³⁸²⁰ - Fixing termination detection on one locality
- Issue #2239³⁸²¹ - Create a new facility lcos::split_all
- Issue #2236³⁸²² - hpx::cout vs. std::cout
- PR #2232³⁸²³ - Implement local-only primary namespace service
- Issue #2147³⁸²⁴ - would like to know how much data is being routed by particular actions
- Issue #2109³⁸²⁵ - Warning while compiling hpx
- Issue #1973³⁸²⁶ - Setting INTERFACE_COMPILE_OPTIONS for hpx_init in CMake taints Fortran_FLAGS
- Issue #1864³⁸²⁷ - run_guarded using bound function ignores reference
- Issue #1754³⁸²⁸ - Running with TCP parcelport causes immediate crash or freeze
- Issue #1655³⁸²⁹ - Enable zip_iterator to be used with Boost traversal iterator categories
- Issue #1591³⁸³⁰ - Optimize AGAS for shared memory only operation
- Issue #1401³⁸³¹ - Need an efficient infiniband parcelport
- Issue #1125³⁸³² - Fix the IPC parcelport
- Issue #839³⁸³³ - Refactor ibverbs and shmem parcelport
- Issue #702³⁸³⁴ - Add instrumentation of parcel layer

³⁸¹² <https://github.com/STELLAR-GROUP/hpx/pull/2251>

³⁸¹³ <https://github.com/STELLAR-GROUP/hpx/pull/2249>

³⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2248>

³⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2246>

³⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2245>

³⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2243>

³⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2242>

³⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2241>

³⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2240>

³⁸²¹ <https://github.com/STELLAR-GROUP/hpx/issues/2239>

³⁸²² <https://github.com/STELLAR-GROUP/hpx/issues/2236>

³⁸²³ <https://github.com/STELLAR-GROUP/hpx/pull/2232>

³⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2147>

³⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2109>

³⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1973>

³⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1864>

³⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1754>

3829 <https://github.com/STELLAR-GROUP/hpx/issues/1655>3830 <https://github.com/STELLAR-GROUP/hpx/issues/1591>3831 <https://github.com/STELLAR-GROUP/hpx/issues/1401>3832 <https://github.com/STELLAR-GROUP/hpx/issues/1125>3833 <https://github.com/STELLAR-GROUP/hpx/issues/839>3834 <https://github.com/STELLAR-GROUP/hpx/issues/702>

- Issue #668³⁸³⁵ - Implement ispc task interface
- Issue #533³⁸³⁶ - Thread queue/deque internal parameters should be runtime configurable
- Issue #475³⁸³⁷ - Create a means of combining performance counters into querysets

2.10.16 HPX V0.9.99 (Jul 15, 2016)

General changes

As the version number of this release hints, we consider this release to be a preview for the upcoming *HPX* V1.0. All of the functionalities we set out to implement for V1.0 are in place; all of the features we wanted to have exposed are ready. We are very happy with the stability and performance of *HPX* and we would like to present this release to the community in order for us to gather broad feedback before releasing V1.0. We still expect for some minor details to change, but on the whole this release represents what we would like to have in a V1.0.

Overall, since the last release we have had almost 1600 commits while closing almost 400 tickets. These numbers reflect the incredible development activity we have seen over the last couple of months. We would like to express a big ‘Thank you!’ to all contributors and those who helped to make this release happen.

The most notable addition in terms of new functionality available with this release is the full implementation of object migration (i.e. the ability to transparently move *HPX* components to a different compute node). Additionally, this release of *HPX* cleans up many minor issues and some API inconsistencies.

Here are some of the main highlights and changes for this release (in no particular order):

- We have fixed a couple of issues in AGAS and the parcel layer which have caused hangs, segmentation faults at exit, and a slowdown of applications over time. Fixing those has significantly increased the overall stability and performance of distributed runs.
- We have started to add parallel algorithm overloads based on the C++ Extensions for Ranges (N4560³⁸³⁸) proposal. This also includes the addition of projections to the existing algorithms. Please see Issue #1668³⁸³⁹ for a list of algorithms which have been adapted to N4560³⁸⁴⁰.
- We have implemented index-based parallel for-loops based on a corresponding standardization proposal (P0075R1³⁸⁴¹). Please see Issue #2016³⁸⁴² for a list of available algorithms.
- We have added implementations for more parallel algorithms as proposed for the upcoming C++ 17 Standard. See Issue #1141³⁸⁴³ 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.

³⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/668>

³⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/533>

³⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/475>

³⁸³⁸ <http://wg21.link/n4560>

³⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1668>

³⁸⁴⁰ <http://wg21.link/n4560>

³⁸⁴¹ <http://wg21.link/p0075r1>

³⁸⁴² <https://github.com/STELLAR-GROUP/hpx/issues/2016>

³⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1141>

- 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 #559³⁸⁴⁴](#) and [PR #1966³⁸⁴⁵](#) 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 #1265³⁸⁴⁶](#) and [PR #1912³⁸⁴⁷](#) for more information.
- Adding overloads of `gather_here` and `gather_there` to accept the plain values of the data to gather (in addition to the existing overloads expecting futures).
- We have cleaned up and refactored large parts of the code base. This helped reducing compile and link times of *HPX* itself and also of applications depending on it. We have further decreased the dependency of *HPX* on the Boost libraries by replacing part of those with facilities available from the standard libraries.
- Wherever possible we have removed dependencies of our API on Boost by replacing those with the equivalent facility from the C++11 standard library.
- We have added new performance counters for parcel coalescing, file-IO, the AGAS cache, and overall scheduler time. Resetting performance counters has been overhauled and fixed.
- We have introduced a generic client type `hpx::components::client<>` and added support for using it with `hpx::async`. This removes the necessity to implement specific client types for every component type without losing type safety. This deemphasizes the need for using the low level `hpx::id_type` for referencing (possibly remote) component instances. The plan is to deprecate the direct use of `hpx::id_type` in user code in the future.
- We have added a special iterator which supports automatic prefetching of one or more arrays for speeding up loop-like code (see `hpx::parallel::util::make_prefetcher_context()`).
- We have extended the interfaces exposed from executors (as proposed by [N4406³⁸⁴⁸](#)) 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 [CMake³⁸⁴⁹](#) 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 ([N4560³⁸⁵⁰](#)) proposal imposes some breaking changes related to the return types of some of the parallel algorithms. Please see [Issue #1668³⁸⁵¹](#) 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>()`.

³⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/559>

³⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1966>

³⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1265>

³⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1912>

³⁸⁴⁸ <http://wg21.link/n4406>

³⁸⁴⁹ <https://www.cmake.org>

³⁸⁵⁰ <http://wg21.link/n4560>

³⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1668>

- 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).

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2250³⁸⁵² - change default chunker of parallel executor to static one
- PR #2247³⁸⁵³ - HPX on ppc64le
- PR #2244³⁸⁵⁴ - Fixing MSVC problems
- PR #2238³⁸⁵⁵ - Fixing small typos
- PR #2237³⁸⁵⁶ - Fixing small typos
- PR #2234³⁸⁵⁷ - Fix broken add test macro when extra args are passed in
- PR #2231³⁸⁵⁸ - Fixing possible race during future awaiting in serialization
- PR #2230³⁸⁵⁹ - Fix stream nvcc
- PR #2229³⁸⁶⁰ - Fixed run_as_hpx_thread
- PR #2228³⁸⁶¹ - On prefetching_test branch : adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2227³⁸⁶² - Support for HPXCL's opencl::event
- PR #2226³⁸⁶³ - Preparing for release of V0.9.99
- PR #2225³⁸⁶⁴ - fix issue when compiling components with hpxcxx
- PR #2224³⁸⁶⁵ - Compute alloc fix

³⁸⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2250>

³⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2247>

³⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2244>

³⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2238>

³⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2237>

³⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2234>

³⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2231>

³⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2230>

³⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2229>

³⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/2228>

³⁸⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2227>

³⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2226>

³⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2225>

³⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2224>

- PR #2223³⁸⁶⁶ - Simplify promise
- PR #2222³⁸⁶⁷ - Replace last uses of boost::function by util::function_nonser
- PR #2221³⁸⁶⁸ - Fix config tests
- PR #2220³⁸⁶⁹ - Fixing gcc 4.6 compilation issues
- PR #2219³⁸⁷⁰ - nullptr support for [unique_] function
- PR #2218³⁸⁷¹ - Introducing clang tidy
- PR #2216³⁸⁷² - Replace NULL with nullptr
- Issue #2214³⁸⁷³ - Let inspect flag use of NULL, suggest nullptr instead
- PR #2213³⁸⁷⁴ - Require support for nullptr
- PR #2212³⁸⁷⁵ - Properly find jemalloc through pkg-config
- PR #2211³⁸⁷⁶ - Disable a couple of warnings reported by Intel on Windows
- PR #2210³⁸⁷⁷ - Fixed host::block_allocator::bulk_construct
- PR #2209³⁸⁷⁸ - Started to clean up new sort algorithms, made things compile for sort_by_key
- PR #2208³⁸⁷⁹ - A couple of fixes that were exposed by a new sort algorithm
- PR #2207³⁸⁸⁰ - Adding missing includes in /hpx/include/serialization.hpp
- PR #2206³⁸⁸¹ - Call package_action::get_future before package_action::apply
- PR #2205³⁸⁸² - The indirect_packaged_task::operator() needs to be run on a HPX thread
- PR #2204³⁸⁸³ - Variadic executor parameters
- PR #2203³⁸⁸⁴ - Delay-initialize members of partitioned iterator
- PR #2202³⁸⁸⁵ - Added segmented fill for hpx::vector
- Issue #2201³⁸⁸⁶ - Null Thread id encountered on partitioned_vector
- PR #2200³⁸⁸⁷ - Fix hangs
- PR #2199³⁸⁸⁸ - Deprecating hpx/traits.hpp

³⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2223>

³⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2222>

³⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2221>

³⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2220>

³⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2219>

³⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2218>

³⁸⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2216>

³⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/2214>

³⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2213>

3875 <https://github.com/STELLAR-GROUP/hpx/pull/2212>3876 <https://github.com/STELLAR-GROUP/hpx/pull/2211>3877 <https://github.com/STELLAR-GROUP/hpx/pull/2210>3878 <https://github.com/STELLAR-GROUP/hpx/pull/2209>3879 <https://github.com/STELLAR-GROUP/hpx/pull/2208>3880 <https://github.com/STELLAR-GROUP/hpx/pull/2207>3881 <https://github.com/STELLAR-GROUP/hpx/pull/2206>3882 <https://github.com/STELLAR-GROUP/hpx/pull/2205>3883 <https://github.com/STELLAR-GROUP/hpx/pull/2204>3884 <https://github.com/STELLAR-GROUP/hpx/pull/2203>3885 <https://github.com/STELLAR-GROUP/hpx/pull/2202>3886 <https://github.com/STELLAR-GROUP/hpx/issues/2201>3887 <https://github.com/STELLAR-GROUP/hpx/pull/2200>3888 <https://github.com/STELLAR-GROUP/hpx/pull/2199>

- PR #2198³⁸⁸⁹ - Making explicit inclusion of external libraries into build
- PR #2197³⁸⁹⁰ - Fix typo in QT CMakeLists
- PR #2196³⁸⁹¹ - Fixing a gcc warning about attributes being ignored
- PR #2194³⁸⁹² - Fixing partitioned_vector_spmd_foreach example
- Issue #2193³⁸⁹³ - partitioned_vector_spmd_foreach seg faults
- PR #2192³⁸⁹⁴ - Support Boost.Thread v4
- PR #2191³⁸⁹⁵ - HPX.Compute prototype
- PR #2190³⁸⁹⁶ - Spawning operation on new thread if remaining stack space becomes too small
- PR #2189³⁸⁹⁷ - Adding callback taking index and future to when_each
- PR #2188³⁸⁹⁸ - Adding new example demonstrating receive_buffer
- PR #2187³⁸⁹⁹ - Mask 128-bit ints if CUDA is being used
- PR #2186³⁹⁰⁰ - Make startup & shutdown functions unique_function
- PR #2185³⁹⁰¹ - Fixing logging output not to cause hang on shutdown
- PR #2184³⁹⁰² - Allowing component clients as action return types
- Issue #2183³⁹⁰³ - Enabling logging output causes hang on shutdown
- Issue #2182³⁹⁰⁴ - 1d_stencil seg fault
- Issue #2181³⁹⁰⁵ - Setting small stack size does not change default
- PR #2180³⁹⁰⁶ - Changing default bind mode to balanced
- PR #2179³⁹⁰⁷ - adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2177³⁹⁰⁸ - Fixing 2176
- Issue #2176³⁹⁰⁹ - Launch process test fails on OSX
- PR #2175³⁹¹⁰ - Fix unbalanced config/warnings includes, add some new ones
- PR #2174³⁹¹¹ - Fix test categorization : regression not unit

³⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2198>

³⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2197>

³⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2196>

³⁸⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2194>

³⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/2193>

³⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2192>

³⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2191>

³⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2190>

³⁸⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2189>

³⁸⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2188>

³⁸⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2187>

³⁹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2186>

³⁹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/2185>

³⁹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2184>

³⁹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/2183>

³⁹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2182>

³⁹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2181>

³⁹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2180>

³⁹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2179>

³⁹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2177>

³⁹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2176>

³⁹¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2175>

³⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/2174>

- Issue #2172³⁹¹² - Different performance results
- Issue #2171³⁹¹³ - “negative entry in reference count table” running octotiger on 32 nodes on queenbee
- Issue #2170³⁹¹⁴ - Error while compiling on Mac + boost 1.60
- PR #2168³⁹¹⁵ - Fixing problems with is_bitwise_serializable
- Issue #2167³⁹¹⁶ - startup & shutdown function should accept unique_function
- Issue #2166³⁹¹⁷ - Simple receive_buffer example
- PR #2165³⁹¹⁸ - Fix wait all
- PR #2164³⁹¹⁹ - Fix wait all
- PR #2163³⁹²⁰ - Fix some typos in config tests
- PR #2162³⁹²¹ - Improve #includes
- PR #2160³⁹²² - Add inspect check for missing #include <list>
- PR #2159³⁹²³ - Add missing finalize call to stop test hanging
- PR #2158³⁹²⁴ - Algo fixes
- PR #2157³⁹²⁵ - Stack check
- Issue #2156³⁹²⁶ - OSX reports stack space incorrectly (generic context coroutines)
- Issue #2155³⁹²⁷ - Race condition suspected in runtime
- PR #2154³⁹²⁸ - Replace boost::detail::atomic_count with the new util::atomic_count
- PR #2153³⁹²⁹ - Fix stack overflow on OSX
- PR #2152³⁹³⁰ - Define is_bitwise_serializable as is_trivially_copyable when available
- PR #2151³⁹³¹ - Adding missing <cstring> for std::mem* functions
- Issue #2150³⁹³² - Unable to use component clients as action return types
- PR #2149³⁹³³ - std::memmove copies bytes, use bytes* sizeof(type) when copying larger types
- PR #2146³⁹³⁴ - Adding customization point for parallel copy/move

³⁹¹² <https://github.com/STELLAR-GROUP/hpx/issues/2172>

³⁹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/2171>

³⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/2170>

³⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2168>

³⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2167>

³⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2166>

³⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2165>

³⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2164>

³⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2163>

³⁹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/2162>

³⁹²² <https://github.com/STELLAR-GROUP/hpx/pull/2160>

³⁹²³ <https://github.com/STELLAR-GROUP/hpx/pull/2159>

³⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2158>

³⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2157>

³⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2156>

³⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2155>

³⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2154>

³⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2153>

³⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2152>

³⁹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/2151>

³⁹³² <https://github.com/STELLAR-GROUP/hpx/issues/2150>

³⁹³³ <https://github.com/STELLAR-GROUP/hpx/pull/2149>

³⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2146>

- PR #2145³⁹³⁵ - Applying changes to address warnings issued by latest version of PVS Studio
- Issue #2148³⁹³⁶ - hpx::parallel::copy is broken after trivially copyable changes
- PR #2144³⁹³⁷ - Some minor tweaks to compute prototype
- PR #2143³⁹³⁸ - Added Boost version support information over OSX platform
- PR #2142³⁹³⁹ - Fixing memory leak in example
- PR #2141³⁹⁴⁰ - Add missing specializations in execution policies
- PR #2139³⁹⁴¹ - This PR fixes a few problems reported by Clang's Undefined Behavior sanitizer
- PR #2138³⁹⁴² - Revert "Adding fedora docs"
- PR #2136³⁹⁴³ - Removed double semicolon
- PR #2135³⁹⁴⁴ - Add deprecated #include check for hpx_fwd.hpp
- PR #2134³⁹⁴⁵ - Resolved memory leak in stencil_8
- PR #2133³⁹⁴⁶ - Replace uses of boost pointer containers
- PR #2132³⁹⁴⁷ - Removing unused typedef
- PR #2131³⁹⁴⁸ - Add several include checks for std facilities
- PR #2130³⁹⁴⁹ - Fixing parcel compression, adding test
- PR #2129³⁹⁵⁰ - Fix invalid attribute warnings
- Issue #2128³⁹⁵¹ - hpx::init seems to segfault
- PR #2127³⁹⁵² - Making executor_traits N-nary
- PR #2126³⁹⁵³ - GCC 4.6 fails to deduce the correct type in lambda
- PR #2125³⁹⁵⁴ - Making parcel coalescing test actually test something
- Issue #2124³⁹⁵⁵ - Make a testcase for parcel compression
- Issue #2123³⁹⁵⁶ - hpx/hpx/runtime/applier_fwd.hpp - Multiple defined types
- Issue #2122³⁹⁵⁷ - Exception in primary_namespace::resolve_free_list

³⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2145>

³⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2148>

³⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2144>

³⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2143>

³⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2142>

³⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2141>

³⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/2139>

³⁹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/2138>

³⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/2136>

³⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2135>

³⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2134>

³⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2133>

³⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2132>

³⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2131>

³⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2130>

³⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2129>

³⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/2128>

³⁹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2127>

³⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2126>

³⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2125>

³⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2124>

³⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2123>

³⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2122>

- Issue #2121³⁹⁵⁸ - Possible memory leak in 1d_stencil_8
- PR #2120³⁹⁵⁹ - Fixing 2119
- Issue #2119³⁹⁶⁰ - reduce_by_key compilation problems
- Issue #2118³⁹⁶¹ - Premature unwrapping of boost::ref'd arguments
- PR #2117³⁹⁶² - Added missing initializer on last constructor for thread_description
- PR #2116³⁹⁶³ - Use a lightweight bind implementation when no placeholders are given
- PR #2115³⁹⁶⁴ - Replace boost::shared_ptr with std::shared_ptr
- PR #2114³⁹⁶⁵ - Adding hook functions for executor_parameter_traits supporting timers
- Issue #2113³⁹⁶⁶ - Compilation error with gcc version 4.9.3 (MacPorts gcc49 4.9.3_0)
- PR #2112³⁹⁶⁷ - Replace uses of safe_bool with explicit operator bool
- Issue #2111³⁹⁶⁸ - Compilation error on QT example
- Issue #2110³⁹⁶⁹ - Compilation error when passing non-future argument to unwrapped continuation in dataflow
- Issue #2109³⁹⁷⁰ - Warning while compiling hpx
- Issue #2109³⁹⁷¹ - Stack trace of last bug causing issues with octotiger
- Issue #2108³⁹⁷² - Stack trace of last bug causing issues with octotiger
- PR #2107³⁹⁷³ - Making sure that a missing parcel_coalescing module does not cause startup exceptions
- PR #2106³⁹⁷⁴ - Stop using hpx_fwd.hpp
- Issue #2105³⁹⁷⁵ - coalescing plugin handler is not optional any more
- Issue #2104³⁹⁷⁶ - Make executor_traits N-nary
- Issue #2103³⁹⁷⁷ - Build error with octotiger and hpx commit e657426d
- PR #2102³⁹⁷⁸ - Combining thread data storage
- PR #2101³⁹⁷⁹ - Added repartition version of 1d stencil that uses any performance counter
- PR #2100³⁹⁸⁰ - Drop obsolete TR1 result_of protocol

³⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2121>

³⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2120>

³⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2119>

³⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2118>

³⁹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/2117>

³⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2116>

³⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2115>

³⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2114>

³⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2113>

³⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2112>

³⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2111>

³⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2110>

³⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2109>

³⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/2109>

³⁹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/2108>

³⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2107>

³⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2106>

³⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2105>

³⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2104>

³⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2103>

³⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2102>

³⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2101>

³⁹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2100>

- PR #2099³⁹⁸¹ - Replace uses of boost::bind with util::bind
- PR #2098³⁹⁸² - Deprecated inspect checks
- PR #2097³⁹⁸³ - Reduce by key, extends #1141
- PR #2096³⁹⁸⁴ - Moving local cache from external to hpx/util
- PR #2095³⁹⁸⁵ - Bump minimum required Boost to 1.50.0
- PR #2094³⁹⁸⁶ - Add include checks for several Boost utilities
- Issue #2093³⁹⁸⁷ - ../../local_cache.hpp(89): error #303: explicit type is missing (“int” assumed)
- PR #2091³⁹⁸⁸ - Fix for Raspberry pi build
- PR #2090³⁹⁸⁹ - Fix storage size for util::function<>
- PR #2089³⁹⁹⁰ - Fix #2088
- Issue #2088³⁹⁹¹ - More verbose output from cmake configuration
- PR #2087³⁹⁹² - Making sure init_globally always executes hpx_main
- Issue #2086³⁹⁹³ - Race condition with recent HPX
- PR #2085³⁹⁹⁴ - Adding #include checker
- PR #2084³⁹⁹⁵ - Replace boost lock types with standard library ones
- PR #2083³⁹⁹⁶ - Simplify packaged task
- PR #2082³⁹⁹⁷ - Updating APEX version for testing
- PR #2081³⁹⁹⁸ - Cleanup exception headers
- PR #2080³⁹⁹⁹ - Make call_once variadic
- Issue #2079⁴⁰⁰⁰ - With GNU C++, line 85 of hpx/config/version.hpp causes link failure when linking application
- Issue #2078⁴⁰⁰¹ - Simple test fails with _GLIBCXX_DEBUG defined
- PR #2077⁴⁰⁰² - Instantiate board in nqueen client
- PR #2076⁴⁰⁰³ - Moving coalescing registration to TUs

³⁹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/2099>

³⁹⁸² <https://github.com/STELLAR-GROUP/hpx/pull/2098>

³⁹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/2097>

³⁹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2096>

³⁹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2095>

³⁹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2094>

³⁹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2093>

³⁹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2091>

³⁹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2090>

³⁹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2089>

³⁹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2088>

³⁹⁹² <https://github.com/STELLAR-GROUP/hpx/pull/2087>

³⁹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/2086>

³⁹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2085>

³⁹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2084>

³⁹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2083>

³⁹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2082>

³⁹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2081>

³⁹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2080>

⁴⁰⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/2079>

⁴⁰⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/2078>

⁴⁰⁰² <https://github.com/STELLAR-GROUP/hpx/pull/2077>

⁴⁰⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/2076>

- PR #2075⁴⁰⁰⁴ - Fixed some documentation typos
- PR #2074⁴⁰⁰⁵ - Adding flush-mode to message handler flush
- PR #2073⁴⁰⁰⁶ - Fixing performance regression introduced lately
- PR #2072⁴⁰⁰⁷ - Refactor local::condition_variable
- PR #2071⁴⁰⁰⁸ - Timer based on boost::asio::deadline_timer
- PR #2070⁴⁰⁰⁹ - Refactor tuple based functionality
- PR #2069⁴⁰¹⁰ - Fixed typos
- Issue #2068⁴⁰¹¹ - Seg fault with octotiger
- PR #2067⁴⁰¹² - Algorithm cleanup
- PR #2066⁴⁰¹³ - Split credit fixes
- PR #2065⁴⁰¹⁴ - Rename HPX_MOVABLE_BUT_NOT_COPYABLE to HPX_MOVABLE_ONLY
- PR #2064⁴⁰¹⁵ - Fixed some typos in docs
- PR #2063⁴⁰¹⁶ - Adding example demonstrating template components
- Issue #2062⁴⁰¹⁷ - Support component templates
- PR #2061⁴⁰¹⁸ - Replace some uses of lexical_cast<string> with C++11 std::to_string
- PR #2060⁴⁰¹⁹ - Replace uses of boost::noncopyable with HPX_NON_COPYABLE
- PR #2059⁴⁰²⁰ - Adding missing for_loop algorithms
- PR #2058⁴⁰²¹ - Move several definitions to more appropriate headers
- PR #2057⁴⁰²² - Simplify assert_owns_lock and ignore_while_checking
- PR #2056⁴⁰²³ - Replacing std::result_of with util::result_of
- PR #2055⁴⁰²⁴ - Fix process launching/connecting back
- PR #2054⁴⁰²⁵ - Add a forwarding coroutine header
- PR #2053⁴⁰²⁶ - Replace uses of boost::unordered_map with std::unordered_map

⁴⁰⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2075>

⁴⁰⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2074>

⁴⁰⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2073>

⁴⁰⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2072>

⁴⁰⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2071>

⁴⁰⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2070>

⁴⁰¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2069>

⁴⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/2068>

⁴⁰¹² <https://github.com/STELLAR-GROUP/hpx/pull/2067>

4013 <https://github.com/STELLAR-GROUP/hpx/pull/2066>4014 <https://github.com/STELLAR-GROUP/hpx/pull/2065>4015 <https://github.com/STELLAR-GROUP/hpx/pull/2064>4016 <https://github.com/STELLAR-GROUP/hpx/pull/2063>4017 <https://github.com/STELLAR-GROUP/hpx/issues/2062>4018 <https://github.com/STELLAR-GROUP/hpx/pull/2061>4019 <https://github.com/STELLAR-GROUP/hpx/pull/2060>4020 <https://github.com/STELLAR-GROUP/hpx/pull/2059>4021 <https://github.com/STELLAR-GROUP/hpx/pull/2058>4022 <https://github.com/STELLAR-GROUP/hpx/pull/2057>4023 <https://github.com/STELLAR-GROUP/hpx/pull/2056>4024 <https://github.com/STELLAR-GROUP/hpx/pull/2055>4025 <https://github.com/STELLAR-GROUP/hpx/pull/2054>4026 <https://github.com/STELLAR-GROUP/hpx/pull/2053>

- PR #2052⁴⁰²⁷ - Rewrite tuple unwrap
- PR #2050⁴⁰²⁸ - Replace uses of BOOST_SCOPED_ENUM with C++11 scoped enums
- PR #2049⁴⁰²⁹ - Attempt to narrow down split_credit problem
- PR #2048⁴⁰³⁰ - Fixing gcc startup hangs
- PR #2047⁴⁰³¹ - Fixing when_xxx and wait_xxx for MSVC12
- PR #2046⁴⁰³² - adding persistent_auto_chunk_size and related tests for for_each
- PR #2045⁴⁰³³ - Fixing HPX_HAVE_THREAD_BACKTRACE_DEPTH build time configuration
- PR #2044⁴⁰³⁴ - Adding missing service executor types
- PR #2043⁴⁰³⁵ - Removing ambiguous definitions for is_future_range and future_range_traits
- PR #2042⁴⁰³⁶ - Clarify that HPX builds can use (much) more than 2GB per process
- PR #2041⁴⁰³⁷ - Changing future_iterator_traits to support pointers
- Issue #2040⁴⁰³⁸ - Improve documentation memory usage warning?
- PR #2039⁴⁰³⁹ - Coroutine cleanup
- PR #2038⁴⁰⁴⁰ - Fix cmake policy CMP0042 warning MACOSX_RPATH
- PR #2037⁴⁰⁴¹ - Avoid redundant specialization of [unique]function_nonser
- PR #2036⁴⁰⁴² - nvcc dies with an internal error upon pushing/popping warnings inside templates
- Issue #2035⁴⁰⁴³ - Use a less restrictive iterator definition in hpx::lcos::detail::future_iterator_traits
- PR #2034⁴⁰⁴⁴ - Fixing compilation error with thread queue wait time performance counter
- Issue #2033⁴⁰⁴⁵ - Compilation error when compiling with thread queue waittime performance counter
- Issue #2032⁴⁰⁴⁶ - Ambiguous template instantiation for is_future_range and future_range_traits.
- PR #2031⁴⁰⁴⁷ - Don't restart timer on every incoming parcel
- PR #2030⁴⁰⁴⁸ - Unify handling of execution policies in parallel algorithms
- PR #2029⁴⁰⁴⁹ - Make pkg-config .pc files use .dylib on OSX

4027 <https://github.com/STELLAR-GROUP/hpx/pull/2052>

4028 <https://github.com/STELLAR-GROUP/hpx/pull/2050>

4029 <https://github.com/STELLAR-GROUP/hpx/pull/2049>

4030 <https://github.com/STELLAR-GROUP/hpx/pull/2048>

4031 <https://github.com/STELLAR-GROUP/hpx/pull/2047>

4032 <https://github.com/STELLAR-GROUP/hpx/pull/2046>

4033 <https://github.com/STELLAR-GROUP/hpx/pull/2045>

4034 <https://github.com/STELLAR-GROUP/hpx/pull/2044>

4035 <https://github.com/STELLAR-GROUP/hpx/pull/2043>

4036 <https://github.com/STELLAR-GROUP/hpx/pull/2042>

4037 <https://github.com/STELLAR-GROUP/hpx/pull/2041>

4038 <https://github.com/STELLAR-GROUP/hpx/issues/2040>

4039 <https://github.com/STELLAR-GROUP/hpx/pull/2039>

4040 <https://github.com/STELLAR-GROUP/hpx/pull/2038>

4041 <https://github.com/STELLAR-GROUP/hpx/pull/2037>

4042 <https://github.com/STELLAR-GROUP/hpx/pull/2036>

4043 <https://github.com/STELLAR-GROUP/hpx/issues/2035>

4044 <https://github.com/STELLAR-GROUP/hpx/pull/2034>

4045 <https://github.com/STELLAR-GROUP/hpx/issues/2033>

4046 <https://github.com/STELLAR-GROUP/hpx/issues/2032>

4047 <https://github.com/STELLAR-GROUP/hpx/pull/2031>

4048 <https://github.com/STELLAR-GROUP/hpx/pull/2030>

4049 <https://github.com/STELLAR-GROUP/hpx/pull/2029>

- PR #2028⁴⁰⁵⁰ - Adding process component
- PR #2027⁴⁰⁵¹ - Making check for compiler compatibility independent on compiler path
- PR #2025⁴⁰⁵² - Fixing inspect tool
- PR #2024⁴⁰⁵³ - Intel13 removal
- PR #2023⁴⁰⁵⁴ - Fix errors related to older boost versions and parameter pack expansions in lambdas
- Issue #2022⁴⁰⁵⁵ - gmake fail: “No rule to make target /usr/lib46/libboost_context-mt.so”
- PR #2021⁴⁰⁵⁶ - Added Sudoku example
- Issue #2020⁴⁰⁵⁷ - Make errors related to init_globally.cpp example while building HPX out of the box
- PR #2019⁴⁰⁵⁸ - Fixed some compilation and cmake errors encountered in nqueen example
- PR #2018⁴⁰⁵⁹ - For loop algorithms
- PR #2017⁴⁰⁶⁰ - Non-recursive at_index implementation
- Issue #2016⁴⁰⁶¹ - Add index-based for-loops
- Issue #2015⁴⁰⁶² - Change default bind-mode to balanced
- PR #2014⁴⁰⁶³ - Fixed dataflow if invoked action returns a future
- PR #2013⁴⁰⁶⁴ - Fixing compilation issues with external example
- PR #2012⁴⁰⁶⁵ - Added Sierpinski Triangle example
- Issue #2011⁴⁰⁶⁶ - Compilation error while running sample hello_world_component code
- PR #2010⁴⁰⁶⁷ - Segmented move implemented for hpx::vector
- Issue #2009⁴⁰⁶⁸ - pkg-config order incorrect on 14.04 / GCC 4.8
- Issue #2008⁴⁰⁶⁹ - Compilation error in dataflow of action returning a future
- PR #2007⁴⁰⁷⁰ - Adding new performance counter exposing overall scheduler time
- PR #2006⁴⁰⁷¹ - Function includes
- PR #2005⁴⁰⁷² - Adding an example demonstrating how to initialize HPX from a global object

⁴⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2028>

⁴⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/2027>

⁴⁰⁵² <https://github.com/STELLAR-GROUP/hpx/pull/2025>

⁴⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/2024>

⁴⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2023>

⁴⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/2022>

⁴⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/2021>

⁴⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2020>

⁴⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/2019>

⁴⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/2018>

⁴⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2017>

⁴⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/2016>

⁴⁰⁶² <https://github.com/STELLAR-GROUP/hpx/issues/2015>

⁴⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/2014>

⁴⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2013>

⁴⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2012>

⁴⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2011>

⁴⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/2010>

⁴⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/2009>

⁴⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/2008>

⁴⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/2007>

⁴⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/2006>

⁴⁰⁷² <https://github.com/STELLAR-GROUP/hpx/pull/2005>

- PR #2004⁴⁰⁷³ - Fixing 2000
- PR #2003⁴⁰⁷⁴ - Adding generation parameter to gather to enable using it more than once
- PR #2002⁴⁰⁷⁵ - Turn on position independent code to solve link problem with hpx_init
- Issue #2001⁴⁰⁷⁶ - Gathering more than once segfaults
- Issue #2000⁴⁰⁷⁷ - Undefined reference to hpx::assertion_failed
- Issue #1999⁴⁰⁷⁸ - Seg fault in hpx::lcos::base_lco_with_value<*>::set_value_nonvirt() when running octo-tiger
- PR #1998⁴⁰⁷⁹ - Detect unknown command line options
- PR #1997⁴⁰⁸⁰ - Extending thread description
- PR #1996⁴⁰⁸¹ - Adding natvis files to solution (MSVC only)
- Issue #1995⁴⁰⁸² - Command line handling does not produce error
- PR #1994⁴⁰⁸³ - Possible missing include in test_utils.hpp
- PR #1993⁴⁰⁸⁴ - Add missing LANGUAGES tag to a hpx_add_compile_flag_if_available() call in CMakeLists.txt
- PR #1992⁴⁰⁸⁵ - Fixing shared_executor_test
- PR #1991⁴⁰⁸⁶ - Making sure the winsock library is properly initialized
- PR #1990⁴⁰⁸⁷ - Fixing bind_test placeholder ambiguity coming from boost-1.60
- PR #1989⁴⁰⁸⁸ - Performance tuning
- PR #1987⁴⁰⁸⁹ - Make configurable size of internal storage in util::function
- PR #1986⁴⁰⁹⁰ - AGAS Refactoring+1753 Cache mods
- PR #1985⁴⁰⁹¹ - Adding missing task_block::run() overload taking an executor
- PR #1984⁴⁰⁹² - Adding an optimized LRU Cache implementation (for AGAS)
- PR #1983⁴⁰⁹³ - Avoid invoking migration table look up for all objects
- PR #1981⁴⁰⁹⁴ - Replacing uintptr_t (which is not defined everywhere) with std::size_t
- PR #1980⁴⁰⁹⁵ - Optimizing LCO continuations

⁴⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/2004>

⁴⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/2003>

⁴⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/2002>

⁴⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/2001>

⁴⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/2000>

⁴⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1999>

⁴⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1998>

⁴⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1997>

⁴⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1996>

⁴⁰⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1995>

⁴⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1994>

⁴⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1993>

⁴⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1992>

⁴⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1991>

⁴⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1990>

⁴⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1989>

⁴⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1987>

⁴⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1986>

⁴⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1985>

⁴⁰⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1984>

⁴⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1983>

⁴⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1981>

⁴⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1980>

- PR #1979⁴⁰⁹⁶ - Fixing Cori
- PR #1978⁴⁰⁹⁷ - Fix test check that got broken in hasty fix to memory overflow
- PR #1977⁴⁰⁹⁸ - Refactor action traits
- PR #1976⁴⁰⁹⁹ - Fixes typo in README.rst
- PR #1975⁴¹⁰⁰ - Reduce size of benchmark timing arrays to fix test failures
- PR #1974⁴¹⁰¹ - Add action to update data owned by the partitioned_vector component
- PR #1972⁴¹⁰² - Adding partitioned_vector SPMD example
- PR #1971⁴¹⁰³ - Fixing 1965
- PR #1970⁴¹⁰⁴ - Papi fixes
- PR #1969⁴¹⁰⁵ - Fixing continuation recursions to not depend on fixed amount of recursions
- PR #1968⁴¹⁰⁶ - More segmented algorithms
- Issue #1967⁴¹⁰⁷ - Simplify component implementations
- PR #1966⁴¹⁰⁸ - Migrate components
- Issue #1964⁴¹⁰⁹ - fatal error: ‘boost/lockfree/detail/branch_hints.hpp’ file not found
- Issue #1962⁴¹¹⁰ - parallel:copy_if has race condition when used on in place arrays
- PR #1963⁴¹¹¹ - Fixing Static Parcelport initialization
- PR #1961⁴¹¹² - Fix function target
- Issue #1960⁴¹¹³ - Papi counters don’t reset
- PR #1959⁴¹¹⁴ - Fixing 1958
- Issue #1958⁴¹¹⁵ - inclusive_scan gives incorrect results with non-commutative operator
- PR #1957⁴¹¹⁶ - Fixing #1950
- PR #1956⁴¹¹⁷ - Sort by key example
- PR #1955⁴¹¹⁸ - Adding regression test for #1946: Hang in wait_all() in distributed run

⁴⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1979>

⁴⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1978>

⁴⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1977>

⁴⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1976>

⁴¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1975>

⁴¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1974>

⁴¹⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1972>

⁴¹⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1971>

⁴¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1970>

⁴¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1969>

⁴¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1968>

⁴¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1967>

⁴¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1966>

⁴¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1964>

4110 <https://github.com/STELLAR-GROUP/hpx/issues/1962>4111 <https://github.com/STELLAR-GROUP/hpx/pull/1963>4112 <https://github.com/STELLAR-GROUP/hpx/pull/1961>4113 <https://github.com/STELLAR-GROUP/hpx/issues/1960>4114 <https://github.com/STELLAR-GROUP/hpx/pull/1959>4115 <https://github.com/STELLAR-GROUP/hpx/issues/1958>4116 <https://github.com/STELLAR-GROUP/hpx/pull/1957>4117 <https://github.com/STELLAR-GROUP/hpx/pull/1956>4118 <https://github.com/STELLAR-GROUP/hpx/pull/1955>

- Issue #1954⁴¹¹⁹ - HPX releases should not use -Werror
- PR #1953⁴¹²⁰ - Adding performance analysis for AGAS cache
- PR #1952⁴¹²¹ - Adapting test for explicit variadics to fail for gcc 4.6
- PR #1951⁴¹²² - Fixing memory leak
- Issue #1950⁴¹²³ - Simplify external builds
- PR #1949⁴¹²⁴ - Fixing yet another lock that is being held during suspension
- PR #1948⁴¹²⁵ - Fixed container algorithms for Intel
- PR #1947⁴¹²⁶ - Adding workaround for tagged_tuple
- Issue #1946⁴¹²⁷ - Hang in wait_all() in distributed run
- PR #1945⁴¹²⁸ - Fixed container algorithm tests
- Issue #1944⁴¹²⁹ - assertion ‘p.destination_locality() == hpx::get_locality()’ failed
- PR #1943⁴¹³⁰ - Fix a couple of compile errors with clang
- PR #1942⁴¹³¹ - Making parcel coalescing functional
- Issue #1941⁴¹³² - Re-enable parcel coalescing
- PR #1940⁴¹³³ - Touching up make_future
- PR #1939⁴¹³⁴ - Fixing problems in over-subscription management in the resource manager
- PR #1938⁴¹³⁵ - Removing use of unified Boost.Thread header
- PR #1937⁴¹³⁶ - Cleaning up the use of Boost.Accumulator headers
- PR #1936⁴¹³⁷ - Making sure interval timer is started for aggregating performance counters
- PR #1935⁴¹³⁸ - Tagged results
- PR #1934⁴¹³⁹ - Fix remote async with deferred launch policy
- Issue #1933⁴¹⁴⁰ - Floating point exception in statistics_counter<boost::accumulators::tag::mean>::get_counter_v
- PR #1932⁴¹⁴¹ - Removing superfluous includes of boost/lockfree/detail/branch_hints.hpp

⁴¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1954>

⁴¹²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1953>

⁴¹²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1952>

⁴¹²² <https://github.com/STELLAR-GROUP/hpx/pull/1951>

⁴¹²³ <https://github.com/STELLAR-GROUP/hpx/issues/1950>

⁴¹²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1949>

⁴¹²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1948>

⁴¹²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1947>

⁴¹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1946>

⁴¹²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1945>

⁴¹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1944>

⁴¹³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1943>

⁴¹³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1942>

⁴¹³² <https://github.com/STELLAR-GROUP/hpx/issues/1941>

⁴¹³³ <https://github.com/STELLAR-GROUP/hpx/pull/1940>

⁴¹³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1939>

⁴¹³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1938>

⁴¹³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1937>

⁴¹³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1936>

⁴¹³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1935>

⁴¹³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1934>

⁴¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1933>

⁴¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1932>

- PR #1931⁴¹⁴² - fix compilation with clang 3.8.0
- Issue #1930⁴¹⁴³ - Missing online documentation for HPX 0.9.11
- PR #1929⁴¹⁴⁴ - LWG2485: get() should be overloaded for const tuple&&
- PR #1928⁴¹⁴⁵ - Revert “Using ninja for circle-ci builds”
- PR #1927⁴¹⁴⁶ - Using ninja for circle-ci builds
- PR #1926⁴¹⁴⁷ - Fixing serialization of std::array
- Issue #1925⁴¹⁴⁸ - Issues with static HPX libraries
- Issue #1924⁴¹⁴⁹ - Performance degrading over time
- Issue #1923⁴¹⁵⁰ - serialization of std::array appears broken in latest commit
- PR #1922⁴¹⁵¹ - Container algorithms
- PR #1921⁴¹⁵² - Tons of smaller quality improvements
- Issue #1920⁴¹⁵³ - Seg fault in hpx::serialization::output_archive::add_gid when running octotiger
- Issue #1919⁴¹⁵⁴ - Intel 15 compiler bug preventing HPX build
- PR #1918⁴¹⁵⁵ - Address sanitizer fixes
- PR #1917⁴¹⁵⁶ - Fixing compilation problems of parallel::sort with Intel compilers
- PR #1916⁴¹⁵⁷ - Making sure code compiles if HPX_WITH_HWLOC=Off
- Issue #1915⁴¹⁵⁸ - max_cores undefined if HPX_WITH_HWLOC=Off
- PR #1913⁴¹⁵⁹ - Add utility member functions for partitioned_vector
- PR #1912⁴¹⁶⁰ - Adding support for invoking actions to dataflow
- PR #1911⁴¹⁶¹ - Adding first batch of container algorithms
- PR #1910⁴¹⁶² - Keep cmake_module_path
- PR #1909⁴¹⁶³ - Fix mpirun with pbs
- PR #1908⁴¹⁶⁴ - Changing parallel::sort to return the last iterator as proposed by N4560

⁴¹⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1931>

⁴¹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1930>

⁴¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1929>

⁴¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1928>

⁴¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1927>

⁴¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1926>

⁴¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1925>

⁴¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1924>

⁴¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1923>

⁴¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1922>

⁴¹⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1921>

⁴¹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1920>

⁴¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1919>

⁴¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1918>

⁴¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1917>

⁴¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1916>

⁴¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1915>

⁴¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1913>

⁴¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1912>

⁴¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1911>

⁴¹⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1910>

⁴¹⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1909>

⁴¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1908>

- PR #1907⁴¹⁶⁵ - Adding a minimum version for Open MPI
- PR #1906⁴¹⁶⁶ - Updates to the Release Procedure
- PR #1905⁴¹⁶⁷ - Fixing #1903
- PR #1904⁴¹⁶⁸ - Making sure std containers are cleared before serialization loads data
- Issue #1903⁴¹⁶⁹ - When running octotiger, I get: assertion '(*new_gids_)[gid].size() == 1' failed: HPX(assertion_failure)
- Issue #1902⁴¹⁷⁰ - Immediate crash when running hpx/octotiger with _GLIBCXX_DEBUG defined.
- PR #1901⁴¹⁷¹ - Making non-serializable classes non-serializable
- Issue #1900⁴¹⁷² - Two possible issues with std::list serialization
- PR #1899⁴¹⁷³ - Fixing a problem with credit splitting as revealed by #1898
- Issue #1898⁴¹⁷⁴ - Accessing component from locality where it was not created segfaults
- PR #1897⁴¹⁷⁵ - Changing parallel::sort to return the last iterator as proposed by N4560
- Issue #1896⁴¹⁷⁶ - version 1.0?
- Issue #1895⁴¹⁷⁷ - Warning comment on numa_allocator is not very clear
- PR #1894⁴¹⁷⁸ - Add support for compilers that have thread_local
- PR #1893⁴¹⁷⁹ - Fixing 1890
- PR #1892⁴¹⁸⁰ - Adds typed future_type for executor_traits
- PR #1891⁴¹⁸¹ - Fix wording in certain parallel algorithm docs
- Issue #1890⁴¹⁸² - Invoking papi counters give segfault
- PR #1889⁴¹⁸³ - Fixing problems as reported by clang-check
- PR #1888⁴¹⁸⁴ - WIP parallel is_heap
- PR #1887⁴¹⁸⁵ - Fixed resetting performance counters related to idle-rate, etc
- Issue #1886⁴¹⁸⁶ - Run hpx with qsub does not work
- PR #1885⁴¹⁸⁷ - Warning cleaning pass

⁴¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1907>

⁴¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1906>

⁴¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1905>

⁴¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1904>

⁴¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1903>

⁴¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1902>

⁴¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1901>

⁴¹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1900>

⁴¹⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1899>

⁴¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1898>

⁴¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1897>

⁴¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1896>

⁴¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1895>

⁴¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1894>

⁴¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1893>

⁴¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1892>

⁴¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1891>

⁴¹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1890>

⁴¹⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1889>

⁴¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1888>

⁴¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1887>

⁴¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1886>

⁴¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1885>

- PR #1884⁴¹⁸⁸ - Add missing parallel algorithm header
- PR #1883⁴¹⁸⁹ - Add feature test for thread_local on Clang for TLS
- PR #1882⁴¹⁹⁰ - Fix some redundant qualifiers
- Issue #1881⁴¹⁹¹ - Unable to compile Octotiger using HPX and Intel MPI on SuperMIC
- Issue #1880⁴¹⁹² - clang with libc++ on Linux needs TLS case
- PR #1879⁴¹⁹³ - Doc fixes for #1868
- PR #1878⁴¹⁹⁴ - Simplify functions
- PR #1877⁴¹⁹⁵ - Removing most usage of Boost.Config
- PR #1876⁴¹⁹⁶ - Add missing parallel algorithms to algorithm.hpp
- PR #1875⁴¹⁹⁷ - Simplify callables
- PR #1874⁴¹⁹⁸ - Address long standing FIXME on using std::unique_ptr with incomplete types
- PR #1873⁴¹⁹⁹ - Fixing 1871
- PR #1872⁴²⁰⁰ - Making sure PBS environment uses specified node list even if no PBS_NODEFILE env is available
- Issue #1871⁴²⁰¹ - Fortran checks should be optional
- PR #1870⁴²⁰² - Touch local::mutex
- PR #1869⁴²⁰³ - Documentation refactoring based off #1868
- PR #1867⁴²⁰⁴ - Embrace static_assert
- PR #1866⁴²⁰⁵ - Fix #1803 with documentation refactoring
- PR #1865⁴²⁰⁶ - Setting OUTPUT_NAME as target properties
- PR #1863⁴²⁰⁷ - Use SYSTEM for boost includes
- PR #1862⁴²⁰⁸ - Minor cleanups
- PR #1861⁴²⁰⁹ - Minor Corrections for Release
- PR #1860⁴²¹⁰ - Fixing hpx gdb script

⁴¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1884>

⁴¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1883>

⁴¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1882>

⁴¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1881>

⁴¹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1880>

⁴¹⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1879>

⁴¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1878>

⁴¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1877>

⁴¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1876>

⁴¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1875>

⁴¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1874>

⁴¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1873>

⁴²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1872>

⁴²⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1871>

⁴²⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1870>

⁴²⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1869>

⁴²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1867>

⁴²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1866>

⁴²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1865>

⁴²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1863>

⁴²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1862>

⁴²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1861>

⁴²¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1860>

- Issue #1859⁴²¹¹ - reset_active_counters resets times and thread counts before some of the counters are evaluated
- PR #1858⁴²¹² - Release V0.9.11
- PR #1857⁴²¹³ - removing diskperf example from 9.11 release
- PR #1856⁴²¹⁴ - fix return in packaged_task_base::reset()
- Issue #1842⁴²¹⁵ - Install error: file INSTALL cannot find libhpx_parcel_coalescing.so.0.9.11
- PR #1839⁴²¹⁶ - Adding fedora docs
- PR #1824⁴²¹⁷ - Changing version on master to V0.9.12
- PR #1818⁴²¹⁸ - Fixing #1748
- Issue #1815⁴²¹⁹ - seg fault in AGAS
- Issue #1803⁴²²⁰ - wait_all documentation
- Issue #1796⁴²²¹ - Outdated documentation to be revised
- Issue #1759⁴²²² - glibc munmap_chunk or free(): invalid pointer on SuperMIC
- Issue #1753⁴²²³ - HPX performance degrades with time since execution begins
- Issue #1748⁴²²⁴ - All public HPX headers need to be self contained
- PR #1719⁴²²⁵ - How to build HPX with Visual Studio
- Issue #1684⁴²²⁶ - Race condition when using -hpx:connect?
- PR #1658⁴²²⁷ - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1641⁴²²⁸ - Generic client
- Issue #1632⁴²²⁹ - heartbeat example fails on separate nodes
- PR #1603⁴²³⁰ - Adds preferred namespace check to inspect tool
- Issue #1559⁴²³¹ - Extend inspect tool
- Issue #1523⁴²³² - Remote async with deferred launch policy never executes
- Issue #1472⁴²³³ - Serialization issues

⁴²¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1859>

⁴²¹² <https://github.com/STELLAR-GROUP/hpx/pull/1858>

⁴²¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1857>

⁴²¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1856>

⁴²¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1842>

⁴²¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1839>

⁴²¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1824>

⁴²¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1818>

⁴²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1815>

⁴²²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1803>

⁴²²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1796>

⁴²²² <https://github.com/STELLAR-GROUP/hpx/issues/1759>

⁴²²³ <https://github.com/STELLAR-GROUP/hpx/issues/1753>

⁴²²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1748>

⁴²²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1719>

⁴²²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1684>

⁴²²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1658>

⁴²²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1641>

⁴²²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1632>

⁴²³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1603>

⁴²³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1559>

⁴²³² <https://github.com/STELLAR-GROUP/hpx/issues/1523>

⁴²³³ <https://github.com/STELLAR-GROUP/hpx/issues/1472>

- Issue #1457⁴²³⁴ - Implement N4392: C++ Latches and Barriers
- PR #1444⁴²³⁵ - Enabling usage of moveonly types for component construction
- Issue #1407⁴²³⁶ - The Intel 13 compiler has failing unit tests
- Issue #1405⁴²³⁷ - Allow component constructors to take movable only types
- Issue #1265⁴²³⁸ - Enable dataflow() to be usable with actions
- Issue #1236⁴²³⁹ - NUMA aware allocators
- Issue #802⁴²⁴⁰ - Fix Broken Examples
- Issue #559⁴²⁴¹ - Add hpx::migrate facility
- Issue #449⁴²⁴² - Make actions with template arguments usable and add documentation
- Issue #279⁴²⁴³ - Refactor addressing_service into a base class and two derived classes
- Issue #224⁴²⁴⁴ - Changing thread state metadata is not thread safe
- Issue #55⁴²⁴⁵ - Uniform syntax for enums should be implemented

2.10.17 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.

⁴²³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1457>

⁴²³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1444>

⁴²³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1407>

⁴²³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1405>

⁴²³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1265>

⁴²³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1236>

⁴²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/802>

⁴²⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/559>

4242 <https://github.com/STELLAR-GROUP/hpx/issues/449>4243 <https://github.com/STELLAR-GROUP/hpx/issues/279>4244 <https://github.com/STELLAR-GROUP/hpx/issues/224>4245 <https://github.com/STELLAR-GROUP/hpx/issues/55>

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 CircleCI⁴²⁴⁶ hosted continuous integration service. This gives us constant and immediate feedback on the health of our master branch.
- The compiler configuration subsystem in the build system has been reimplemented. Instead of using Boost.Config we now use our own lightweight set of cmake scripts to determine the available language and library features supported by the used compiler.
- The API for creating instances of components has been consolidated. All component instances should be created using the `hpx::new_` only. It allows one to instantiate both, single component instances and multiple component instances. The placement of the created components can be controlled by special distribution policies. Please see the corresponding documentation outlining the use of `hpx::new_`.
- Introduced four new distribution policies which can be used with many API functions which traditionally expected to be used with a locality id. The new distribution policies are:
 - `hpx::components::default_distribution_policy` which tries to place multiple component instances as evenly as possible.
 - `hpx::components::colocating_distribution_policy` which will refer to the locality where a given component instance is currently placed.
 - `hpx::components::binpacking_distribution_policy` which will place multiple component instances as evenly as possible based on any performance counter.
 - `hpx::components::target_distribution_policy` which allows one to represent a given locality in the context of a 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 N4411⁴²⁴⁷.
- 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 N4399⁴²⁴⁸).
- 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`.

⁴²⁴⁶ <https://circleci.com/gh/STELLAR-GROUP/hpx>

⁴²⁴⁷ <http://wg21.link/n4411>

⁴²⁴⁸ <http://wg21.link/n4399>

- 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 #1141⁴²⁴⁹ for the list of all available parallel algorithms (thanks to Daniel Bourgeois and John Biddiscombe for contributing their work).

Breaking changes

- We are moving into the direction of unifying managed and simple components. In order to stop exposing the old facilities, all examples have been converted to use the new classes. The breaking change in this release is that performance counters are now a `hpx::components::component_base` instead of `hpx::components::managed_component_base`.
- We removed the support for stackless threads. It turned out that there was no performance benefit when using stackless threads. As such, we decided to clean up our codebase. This feature was not documented.
- The CMake project name has changed from ‘hpx’ to ‘HPX’ for consistency and compatibility with naming conventions and other CMake projects. Generated config files go into `<prefix>/lib/cmake/HPX` and not `<prefix>/lib/cmake/hpx`.
- The macro `HPX_REGISTER_MINIMAL_COMPONENT_FACTORY` has been deprecated. Please use `HPX_REGISTER_COMPONENT` instead. The old macro will be removed in the next release.
- The obsolete distributing_factory and binpacking_factory components have been removed. The corresponding functionality is now provided by the `hpx::new_` API function in conjunction with the `hpx::default_layout` and `hpx::binpacking` distribution policies (`hpx::components::default_distribution_policy` and `hpx::components::binpacking_distribution_policy`)
- The API function `hpx::new_colocated` has been deprecated. Please use the consolidated API `hpx::new_` in conjunction with the new `hpx::colocated` distribution policy (`hpx::components::colocating_distribution_policy`) instead. The old API function will still be available for at least one release of `HPX` if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The API function `hpx::async_colocated` has been deprecated. Please use the consolidated API `hpx::async` in conjunction with the new `hpx::colocated` distribution policy (`hpx::components::colocating_distribution_policy`) instead. The old API function will still be available for at least one release of `HPX` if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The obsolete remote_object component has been removed.

⁴²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1141>

- 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 announcement⁴²⁵⁰ 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`.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #1855⁴²⁵¹ - Completely removing external/endian
- PR #1854⁴²⁵² - Don't pollute CMAKE_CXX_FLAGS through `find_package()`
- PR #1853⁴²⁵³ - Updating CMake configuration to get correct version of TAU library
- PR #1852⁴²⁵⁴ - Fixing Performance Problems with MPI Parcelport
- PR #1851⁴²⁵⁵ - Fixing `hpx_add_link_flag()` and `hpx_remove_link_flag()`
- PR #1850⁴²⁵⁶ - Fixing 1836, adding `parallel::sort`
- PR #1849⁴²⁵⁷ - Fixing configuration for use of more than 64 cores
- PR #1848⁴²⁵⁸ - Change default APEX version for release
- PR #1847⁴²⁵⁹ - Fix `client_base::then` on release
- PR #1846⁴²⁶⁰ - Removing broken `lcos::local::channel` from release
- PR #1845⁴²⁶¹ - Adding example demonstrating a possible safe-object implementation to release
- PR #1844⁴²⁶² - Removing stubs from accumulator examples

⁴²⁵⁰ <http://thread.gmane.org/gmane.comp.lib.hpx.devel/196>

⁴²⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1855>

⁴²⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1854>

⁴²⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1853>

⁴²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1852>

⁴²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1851>

⁴²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1850>

⁴²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1849>

⁴²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1848>

⁴²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1847>

⁴²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1846>

⁴²⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1845>

⁴²⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1844>

- PR #1843⁴²⁶³ - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1841⁴²⁶⁴ - Fixing client_base<>::then
- PR #1840⁴²⁶⁵ - Adding example demonstrating a possible safe-object implementation
- PR #1838⁴²⁶⁶ - Update version rc1
- PR #1837⁴²⁶⁷ - Removing broken lcos::local::channel
- PR #1835⁴²⁶⁸ - Adding explicit move constructor and assignment operator to hpx::lcos::promise
- PR #1834⁴²⁶⁹ - Making hpx::lcos::promise move-only
- PR #1833⁴²⁷⁰ - Adding fedora docs
- Issue #1832⁴²⁷¹ - hpx::lcos::promise<> must be move-only
- PR #1831⁴²⁷² - Fixing resource manager gcc5.2
- PR #1830⁴²⁷³ - Fix intel13
- PR #1829⁴²⁷⁴ - Unbreaking thread test
- PR #1828⁴²⁷⁵ - Fixing #1620
- PR #1827⁴²⁷⁶ - Fixing a memory management issue for the Parquet application
- Issue #1826⁴²⁷⁷ - Memory management issue in hpx::lcos::promise
- PR #1825⁴²⁷⁸ - Adding hpx::components::component and hpx::components::component_base
- PR #1823⁴²⁷⁹ - Adding git commit id to circleci build
- PR #1822⁴²⁸⁰ - applying fixes suggested by clang 3.7
- PR #1821⁴²⁸¹ - Hyperlink fixes
- PR #1820⁴²⁸² - added parallel multi-locality sanity test
- PR #1819⁴²⁸³ - Fixing #1667
- Issue #1817⁴²⁸⁴ - Hyperlinks generated by inspect tool are wrong
- PR #1816⁴²⁸⁵ - Support hpxrx

⁴²⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1843>

⁴²⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1841>

⁴²⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1840>

⁴²⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1838>

⁴²⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1837>

⁴²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1835>

⁴²⁶⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1834>

⁴²⁷⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1833>

⁴²⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1832>

⁴²⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1831>

⁴²⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1830>

⁴²⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1829>

⁴²⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1828>

⁴²⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1827>

⁴²⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1826>

⁴²⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1825>

⁴²⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1823>

⁴²⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1822>

⁴²⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1821>

⁴²⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1820>

⁴²⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1819>

⁴²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1817>

⁴²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1816>

- PR #1814⁴²⁸⁶ - Fix async to dispatch to the correct locality in all cases
- Issue #1813⁴²⁸⁷ - `async(launch::..., action(), ...)` always invokes locally
- PR #1812⁴²⁸⁸ - fixed syntax error in CMakeLists.txt
- PR #1811⁴²⁸⁹ - Agas optimizations
- PR #1810⁴²⁹⁰ - drop superfluous typedefs
- PR #1809⁴²⁹¹ - Allow HPX to be used as an optional package in 3rd party code
- PR #1808⁴²⁹² - Fixing #1723
- PR #1807⁴²⁹³ - Making sure resolve_localities does not hang during normal operation
- Issue #1806⁴²⁹⁴ - Spinlock no longer movable and deletes operator '=' , breaks MiniGhost
- Issue #1804⁴²⁹⁵ - register_with_basename causes hangs
- PR #1801⁴²⁹⁶ - Enhanced the inspect tool to take user directly to the problem with hyperlinks
- Issue #1800⁴²⁹⁷ - Problems compiling application on smic
- PR #1799⁴²⁹⁸ - Fixing cv exceptions
- PR #1798⁴²⁹⁹ - Documentation refactoring & updating
- PR #1797⁴³⁰⁰ - Updating the activeharmony CMake module
- PR #1795⁴³⁰¹ - Fixing cv
- PR #1794⁴³⁰² - Fix connect with `hpx::runtime_mode_connect`
- PR #1793⁴³⁰³ - fix a wrong use of `HPX_MAX_CPU_COUNT` instead of `HPX_HAVE_MAX_CPU_COUNT`
- PR #1792⁴³⁰⁴ - Allow for default constructed parcel instances to be moved
- PR #1791⁴³⁰⁵ - Fix connect with `hpx::runtime_mode_connect`
- Issue #1790⁴³⁰⁶ - assertion `action_.get()` failed: `HPX(assertion_failure)` when running Octotiger with pull request 1786
- PR #1789⁴³⁰⁷ - Fixing discover_counter_types API function
- Issue #1788⁴³⁰⁸ - connect with `hpx::runtime_mode_connect`

4286 <https://github.com/STELLAR-GROUP/hpx/pull/1814>

4287 <https://github.com/STELLAR-GROUP/hpx/issues/1813>

4288 <https://github.com/STELLAR-GROUP/hpx/pull/1812>

4289 <https://github.com/STELLAR-GROUP/hpx/pull/1811>

4290 <https://github.com/STELLAR-GROUP/hpx/pull/1810>

4291 <https://github.com/STELLAR-GROUP/hpx/pull/1809>

4292 <https://github.com/STELLAR-GROUP/hpx/pull/1808>

4293 <https://github.com/STELLAR-GROUP/hpx/pull/1807>

4294 <https://github.com/STELLAR-GROUP/hpx/issues/1806>

4295 <https://github.com/STELLAR-GROUP/hpx/issues/1804>

4296 <https://github.com/STELLAR-GROUP/hpx/pull/1801>

4297 <https://github.com/STELLAR-GROUP/hpx/issues/1800>

4298 <https://github.com/STELLAR-GROUP/hpx/pull/1799>

4299 <https://github.com/STELLAR-GROUP/hpx/pull/1798>

4300 <https://github.com/STELLAR-GROUP/hpx/pull/1797>

4301 <https://github.com/STELLAR-GROUP/hpx/pull/1795>

4302 <https://github.com/STELLAR-GROUP/hpx/pull/1794>

4303 <https://github.com/STELLAR-GROUP/hpx/pull/1793>

4304 <https://github.com/STELLAR-GROUP/hpx/pull/1792>

4305 <https://github.com/STELLAR-GROUP/hpx/pull/1791>

4306 <https://github.com/STELLAR-GROUP/hpx/issues/1790>

4307 <https://github.com/STELLAR-GROUP/hpx/pull/1789>

4308 <https://github.com/STELLAR-GROUP/hpx/issues/1788>

- Issue #1787⁴³⁰⁹ - discover_counter_types not working
- PR #1786⁴³¹⁰ - Changing addressing_service to use std::unordered_map instead of std::map
- PR #1785⁴³¹¹ - Fix is_iterator for container algorithms
- PR #1784⁴³¹² - Adding new command line options:
- PR #1783⁴³¹³ - Minor changes for APEX support
- PR #1782⁴³¹⁴ - Drop legacy forwarding action traits
- PR #1781⁴³¹⁵ - Attempt to resolve the race between cv::wait_xxx and cv::notify_all
- PR #1780⁴³¹⁶ - Removing serialize_sequence
- PR #1779⁴³¹⁷ - Fixed #1501: hwloc configuration options are wrong for MIC
- PR #1778⁴³¹⁸ - Removing ability to enable/disable parcel handling
- PR #1777⁴³¹⁹ - Completely removing stackless threads
- PR #1776⁴³²⁰ - Cleaning up util/plugin
- PR #1775⁴³²¹ - Agas fixes
- PR #1774⁴³²² - Action invocation count
- PR #1773⁴³²³ - replaced MSVC variable with WIN32
- PR #1772⁴³²⁴ - Fixing Problems in MPI parcelport and future serialization.
- PR #1771⁴³²⁵ - Fixing intel 13 compiler errors related to variadic template template parameters for lcos::when_tests
- PR #1770⁴³²⁶ - Forwarding decay to std:::
- PR #1769⁴³²⁷ - Add more characters with special regex meaning to the existing patch
- PR #1768⁴³²⁸ - Adding test for receive_buffer
- PR #1767⁴³²⁹ - Making sure that uptime counter throws exception on any attempt to be reset
- PR #1766⁴³³⁰ - Cleaning up code related to throttling scheduler
- PR #1765⁴³³¹ - Restricting thread_data to creating only with intrusive_pointers

⁴³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1787>

⁴³¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1786>

⁴³¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1785>

⁴³¹² <https://github.com/STELLAR-GROUP/hpx/pull/1784>

⁴³¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1783>

⁴³¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1782>

⁴³¹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1781>

⁴³¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1780>

⁴³¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1779>

⁴³¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1778>

⁴³¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1777>

⁴³²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1776>

⁴³²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1775>

⁴³²² <https://github.com/STELLAR-GROUP/hpx/pull/1774>

⁴³²³ <https://github.com/STELLAR-GROUP/hpx/pull/1773>

⁴³²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1772>

⁴³²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1771>

⁴³²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1770>

⁴³²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1769>

⁴³²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1768>

⁴³²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1767>

⁴³³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1766>

⁴³³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1765>

- PR #1764⁴³³² - Fixing 1763
- Issue #1763⁴³³³ - UB in thread_data::operator delete
- PR #1762⁴³³⁴ - Making sure all serialization registries/factories are unique
- PR #1761⁴³³⁵ - Fixed #1751: hpx::future::wait_for fails a simple test
- PR #1758⁴³³⁶ - Fixing #1757
- Issue #1757⁴³³⁷ - pinning not correct using --hpx:bind
- Issue #1756⁴³³⁸ - compilation error with MinGW
- PR #1755⁴³³⁹ - Making output serialization const-correct
- Issue #1753⁴³⁴⁰ - HPX performance degrades with time since execution begins
- Issue #1752⁴³⁴¹ - Error in AGAS
- Issue #1751⁴³⁴² - hpx::future::wait_for fails a simple test
- PR #1750⁴³⁴³ - Removing hpx_fwd.hpp includes
- PR #1749⁴³⁴⁴ - Simplify result_of and friends
- PR #1747⁴³⁴⁵ - Removed superfluous code from message_buffer.hpp
- PR #1746⁴³⁴⁶ - Tuple dependencies
- Issue #1745⁴³⁴⁷ - Broken when_some which takes iterators
- PR #1744⁴³⁴⁸ - Refining archive interface
- PR #1743⁴³⁴⁹ - Fixing when_all when only a single future is passed
- PR #1742⁴³⁵⁰ - Config includes
- PR #1741⁴³⁵¹ - Os executors
- Issue #1740⁴³⁵² - hpx::promise has some problems
- PR #1739⁴³⁵³ - Parallel composition with generic containers
- Issue #1738⁴³⁵⁴ - After building program and successfully linking to a version of hpx DHPX_DIR seems to be ignored

⁴³³² <https://github.com/STELLAR-GROUP/hpx/pull/1764>

⁴³³³ <https://github.com/STELLAR-GROUP/hpx/issues/1763>

⁴³³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1762>

⁴³³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1761>

⁴³³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1758>

⁴³³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1757>

⁴³³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1756>

⁴³³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1755>

⁴³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1753>

⁴³⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1752>

⁴³⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1751>

⁴³⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1750>

⁴³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1749>

⁴³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1747>

⁴³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1746>

⁴³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1745>

⁴³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1744>

⁴³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1743>

⁴³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1742>

⁴³⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1741>

⁴³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1740>

⁴³⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1739>

⁴³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1738>

- Issue #1737⁴³⁵⁵ - Uptime problems
- PR #1736⁴³⁵⁶ - added convenience c-tor and begin()/end() to serialize_buffer
- PR #1735⁴³⁵⁷ - Config includes
- PR #1734⁴³⁵⁸ - Fixed #1688: Add timer counters for tfunc_total and exec_total
- Issue #1733⁴³⁵⁹ - Add unit test for hpx/lcos/local/receive_buffer.hpp
- PR #1732⁴³⁶⁰ - Renaming get_os_thread_count
- PR #1731⁴³⁶¹ - Basename registration
- Issue #1730⁴³⁶² - Use after move of thread_init_data
- PR #1729⁴³⁶³ - Rewriting channel based on new gate component
- PR #1728⁴³⁶⁴ - Fixing #1722
- PR #1727⁴³⁶⁵ - Fixing compile problems with apply_colocated
- PR #1726⁴³⁶⁶ - Apex integration
- PR #1725⁴³⁶⁷ - fixed test timeouts
- PR #1724⁴³⁶⁸ - Renaming vector
- Issue #1723⁴³⁶⁹ - Drop support for intel compilers and gcc 4.4. based standard libs
- Issue #1722⁴³⁷⁰ - Add support for detecting non-ready futures before serialization
- PR #1721⁴³⁷¹ - Unifying parallel executors, initializing from launch policy
- PR #1720⁴³⁷² - dropped superfluous typedef
- Issue #1718⁴³⁷³ - Windows 10 x64, VS 2015 - Unknown CMake command “add_hpx_pseudo_target”.
- PR #1717⁴³⁷⁴ - Timed executor traits for thread-executors
- PR #1716⁴³⁷⁵ - serialization of arrays didn't work with non-pod types. fixed
- PR #1715⁴³⁷⁶ - List serialization
- PR #1714⁴³⁷⁷ - changing misspellings

⁴³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1737>

⁴³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1736>

⁴³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1735>

⁴³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1734>

⁴³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1733>

⁴³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1732>

⁴³⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1731>

⁴³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1730>

⁴³⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1729>

⁴³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1728>

⁴³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1727>

⁴³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1726>

⁴³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1725>

⁴³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1724>

⁴³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1723>

⁴³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1722>

⁴³⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1721>

⁴³⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1720>

⁴³⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1718>

⁴³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1717>

⁴³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1716>

⁴³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1715>

⁴³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1714>

- PR #1713⁴³⁷⁸ - Fixed distribution policy executors
- PR #1712⁴³⁷⁹ - Moving library detection to be executed after feature tests
- PR #1711⁴³⁸⁰ - Simplify parcel
- PR #1710⁴³⁸¹ - Compile only tests
- PR #1709⁴³⁸² - Implemented timed executors
- PR #1708⁴³⁸³ - Implement parallel::executor_traits for thread-executors
- PR #1707⁴³⁸⁴ - Various fixes to threads::executors to make custom schedulers work
- PR #1706⁴³⁸⁵ - Command line option –hpx:cores does not work as expected
- Issue #1705⁴³⁸⁶ - command line option –hpx:cores does not work as expected
- PR #1704⁴³⁸⁷ - vector deserialization is speeded up a little
- PR #1703⁴³⁸⁸ - Fixing shared_mutes
- Issue #1702⁴³⁸⁹ - Shared_mutex does not compile with no_mutex cond_var
- PR #1701⁴³⁹⁰ - Add distribution_policy_executor
- PR #1700⁴³⁹¹ - Executor parameters
- PR #1699⁴³⁹² - Readers writer lock
- PR #1698⁴³⁹³ - Remove leftovers
- PR #1697⁴³⁹⁴ - Fixing held locks
- PR #1696⁴³⁹⁵ - Modified Scan Partitioner for Algorithms
- PR #1695⁴³⁹⁶ - This thread executors
- PR #1694⁴³⁹⁷ - Fixed #1688: Add timer counters for tfunc_total and exec_total
- PR #1693⁴³⁹⁸ - Fix #1691: is_executor template specification fails for inherited executors
- PR #1692⁴³⁹⁹ - Fixed #1662: Possible exception source in coalescing_message_handler
- Issue #1691⁴⁴⁰⁰ - is_executor template specification fails for inherited executors

⁴³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1713>

⁴³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1712>

⁴³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1711>

⁴³⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1710>

⁴³⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1709>

⁴³⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1708>

⁴³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1707>

⁴³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1706>

⁴³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1705>

⁴³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1704>

⁴³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1703>

⁴³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1702>

⁴³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1701>

⁴³⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1700>

⁴³⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1699>

⁴³⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1698>

⁴³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1697>

⁴³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1696>

⁴³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1695>

⁴³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1694>

⁴³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1693>

⁴³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1692>

⁴⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1691>

- PR #1690⁴⁴⁰¹ - added macro for non-intrusive serialization of classes without a default c-tor
- PR #1689⁴⁴⁰² - Replace value_or_error with custom storage, unify future_data state
- Issue #1688⁴⁴⁰³ - Add timer counters for tfunc_total and exec_total
- PR #1687⁴⁴⁰⁴ - Fixed interval timer
- PR #1686⁴⁴⁰⁵ - Fixing cmake warnings about not existing pseudo target dependencies
- PR #1685⁴⁴⁰⁶ - Converting partitioners to use bulk async execute
- PR #1683⁴⁴⁰⁷ - Adds a tool for inspect that checks for character limits
- PR #1682⁴⁴⁰⁸ - Change project name to (uppercase) HPX
- PR #1681⁴⁴⁰⁹ - Counter shortnames
- PR #1680⁴⁴¹⁰ - Extended Non-intrusive Serialization to Ease Usage for Library Developers
- PR #1679⁴⁴¹¹ - Working on 1544: More executor changes
- PR #1678⁴⁴¹² - Transpose fixes
- PR #1677⁴⁴¹³ - Improve Boost compatibility check
- PR #1676⁴⁴¹⁴ - 1d stencil fix
- Issue #1675⁴⁴¹⁵ - hpx project name is not HPX
- PR #1674⁴⁴¹⁶ - Fixing the MPI parcelport
- PR #1673⁴⁴¹⁷ - added move semantics to map/vector deserialization
- PR #1672⁴⁴¹⁸ - Vs2015 await
- PR #1671⁴⁴¹⁹ - Adapt transform for #1668
- PR #1670⁴⁴²⁰ - Started to work on #1668
- PR #1669⁴⁴²¹ - Add this_thread_executors
- Issue #1667⁴⁴²² - Apple build instructions in docs are out of date
- PR #1666⁴⁴²³ - Apex integration

⁴⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1690>

⁴⁴⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1689>

⁴⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1688>

⁴⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1687>

⁴⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1686>

⁴⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1685>

⁴⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1683>

⁴⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1682>

⁴⁴⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1681>

⁴⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1680>

⁴⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1679>

⁴⁴¹² <https://github.com/STELLAR-GROUP/hpx/pull/1678>

⁴⁴¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1677>

⁴⁴¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1676>

⁴⁴¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1675>

⁴⁴¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1674>

⁴⁴¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1673>

⁴⁴¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1672>

⁴⁴¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1671>

⁴⁴²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1670>

⁴⁴²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1669>

⁴⁴²² <https://github.com/STELLAR-GROUP/hpx/issues/1667>

⁴⁴²³ <https://github.com/STELLAR-GROUP/hpx/pull/1666>

- PR #1665⁴⁴²⁴ - Fixes an error with the whitespace check that showed the incorrect location of the error
- Issue #1664⁴⁴²⁵ - Inspect tool found incorrect endline whitespace
- PR #1663⁴⁴²⁶ - Improve use of locks
- Issue #1662⁴⁴²⁷ - Possible exception source in coalescing_message_handler
- PR #1661⁴⁴²⁸ - Added support for 128bit number serialization
- PR #1660⁴⁴²⁹ - Serialization 128bits
- PR #1659⁴⁴³⁰ - Implemented inner_product and adjacent_diff algos
- PR #1658⁴⁴³¹ - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1657⁴⁴³² - Use of shared_ptr in io_service_pool changed to unique_ptr
- Issue #1656⁴⁴³³ - 1d_stencil codes all have wrong factor
- PR #1654⁴⁴³⁴ - When using runtime_mode_connect, find the correct localhost public ip address
- PR #1653⁴⁴³⁵ - Fixing 1617
- PR #1652⁴⁴³⁶ - Remove traits::action_may_require_id_splitting
- PR #1651⁴⁴³⁷ - Fixed performance counters related to AGAS cache timings
- PR #1650⁴⁴³⁸ - Remove leftovers of traits::type_size
- PR #1649⁴⁴³⁹ - Shorten target names on Windows to shorten used path names
- PR #1648⁴⁴⁴⁰ - Fixing problems introduced by merging #1623 for older compilers
- PR #1647⁴⁴⁴¹ - Simplify running automatic builds on Windows
- Issue #1646⁴⁴⁴² - Cache insert and update performance counters are broken
- Issue #1644⁴⁴⁴³ - Remove leftovers of traits::type_size
- Issue #1643⁴⁴⁴⁴ - Remove traits::action_may_require_id_splitting
- PR #1642⁴⁴⁴⁵ - Adds spell checker to the inspect tool for qbk and doxygen comments
- PR #1640⁴⁴⁴⁶ - First step towards fixing 688

⁴⁴²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1665>

⁴⁴²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1664>

⁴⁴²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1663>

⁴⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1662>

⁴⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1661>

⁴⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1660>

⁴⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1659>

⁴⁴³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1658>

⁴⁴³² <https://github.com/STELLAR-GROUP/hpx/pull/1657>

⁴⁴³³ <https://github.com/STELLAR-GROUP/hpx/issues/1656>

⁴⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1654>

⁴⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1653>

⁴⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1652>

⁴⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1651>

⁴⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1650>

⁴⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1649>

⁴⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1648>

⁴⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1647>

⁴⁴⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1646>

⁴⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1644>

⁴⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1643>

⁴⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1642>

⁴⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1640>

- PR #1639⁴⁴⁴⁷ - Re-apply remaining changes from limit_dataflow_recursion branch
- PR #1638⁴⁴⁴⁸ - This fixes possible deadlock in the test ignore_while_locked_1485
- PR #1637⁴⁴⁴⁹ - Fixing hpx::wait_all() invoked with two vector<future<T>>
- PR #1636⁴⁴⁵⁰ - Partially re-apply changes from limit_dataflow_recursion branch
- PR #1635⁴⁴⁵¹ - Adding missing test for #1572
- PR #1634⁴⁴⁵² - Revert “Limit recursion-depth in dataflow to a configurable constant”
- PR #1633⁴⁴⁵³ - Add command line option to ignore batch environment
- PR #1631⁴⁴⁵⁴ - hpx::lcos::queue exhibits strange behavior
- PR #1630⁴⁴⁵⁵ - Fixed endline_whitespace_check.cpp to detect lines with only whitespace
- Issue #1629⁴⁴⁵⁶ - Inspect trailing whitespace checker problem
- PR #1628⁴⁴⁵⁷ - Removed meaningless const qualifiers. Minor icpc fix.
- PR #1627⁴⁴⁵⁸ - Fixing the queue LCO and add example demonstrating its use
- PR #1626⁴⁴⁵⁹ - Deprecating get_gid(), add get_id() and get_unmanaged_id()
- PR #1625⁴⁴⁶⁰ - Allowing to specify whether to send credits along with message
- Issue #1624⁴⁴⁶¹ - Lifetime issue
- Issue #1623⁴⁴⁶² - hpx::wait_all() invoked with two vector<future<T>> fails
- PR #1622⁴⁴⁶³ - Executor partitioners
- PR #1621⁴⁴⁶⁴ - Clean up coroutines implementation
- Issue #1620⁴⁴⁶⁵ - Revert #1535
- PR #1619⁴⁴⁶⁶ - Fix result type calculation for hpx::make_continuation
- PR #1618⁴⁴⁶⁷ - Fixing RDTSC on Xeon/Phi
- Issue #1617⁴⁴⁶⁸ - hpx cmake not working when run as a subproject
- Issue #1616⁴⁴⁶⁹ - cmake problem resulting in RDTSC not working correctly for Xeon Phi creates very strange results for duration counters

⁴⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1639>

⁴⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1638>

⁴⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1637>

⁴⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1636>

⁴⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/pull/1635>

⁴⁴⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1634>

⁴⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1633>

⁴⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1631>

⁴⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1630>

⁴⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1629>

⁴⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1628>

⁴⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1627>

⁴⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1626>

⁴⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1625>

⁴⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1624>

⁴⁴⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1623>

⁴⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1622>

⁴⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1621>

⁴⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1620>

⁴⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1619>

⁴⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1618>

⁴⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1617>

⁴⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1616>

- Issue #1615⁴⁴⁷⁰ - hpx::make_continuation requires input and output to be the same
- PR #1614⁴⁴⁷¹ - Fixed remove copy test
- Issue #1613⁴⁴⁷² - Dataflow causes stack overflow
- PR #1612⁴⁴⁷³ - Modified foreach partitioner to use bulk execute
- PR #1611⁴⁴⁷⁴ - Limit recursion-depth in dataflow to a configurable constant
- PR #1610⁴⁴⁷⁵ - Increase timeout for CircleCI
- PR #1609⁴⁴⁷⁶ - Refactoring thread manager, mainly extracting thread pool
- PR #1608⁴⁴⁷⁷ - Fixed running multiple localities without localities parameter
- PR #1607⁴⁴⁷⁸ - More algorithm fixes to adjacentfind
- Issue #1606⁴⁴⁷⁹ - Running without localities parameter binds to bogus port range
- Issue #1605⁴⁴⁸⁰ - Too many serializations
- PR #1604⁴⁴⁸¹ - Changes the HPX image into a hyperlink
- PR #1601⁴⁴⁸² - Fixing problems with remove_copy algorithm tests
- PR #1600⁴⁴⁸³ - Actions with ids cleanup
- PR #1599⁴⁴⁸⁴ - Duplicate binding of global ids should fail
- PR #1598⁴⁴⁸⁵ - Fixing array access
- PR #1597⁴⁴⁸⁶ - Improved the reliability of connecting/disconnecting localities
- Issue #1596⁴⁴⁸⁷ - Duplicate id binding should fail
- PR #1595⁴⁴⁸⁸ - Fixing more cmake config constants
- PR #1594⁴⁴⁸⁹ - Fixing preprocessor constant used to enable C++11 chrono
- PR #1593⁴⁴⁹⁰ - Adding operator||() for hpx::launch
- Issue #1592⁴⁴⁹¹ - Error (typo) in the docs
- Issue #1590⁴⁴⁹² - CMake fails when CMAKE_BINARY_DIR contains '+'.

⁴⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1615>

⁴⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1614>

⁴⁴⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1613>

⁴⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1612>

⁴⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1611>

⁴⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1610>

⁴⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1609>

⁴⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1608>

⁴⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1607>

⁴⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1606>

⁴⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1605>

⁴⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/pull/1604>

⁴⁴⁸² <https://github.com/STELLAR-GROUP/hpx/pull/1601>

⁴⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/pull/1600>

⁴⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1599>

⁴⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1598>

⁴⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1597>

⁴⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1596>

⁴⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1595>

⁴⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1594>

⁴⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1593>

⁴⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1592>

⁴⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1590>

- Issue #1589⁴⁴⁹³ - Disconnecting a locality results in segfault using heartbeat example
- PR #1588⁴⁴⁹⁴ - Fix doc string for config option HPX_WITH_EXAMPLES
- PR #1586⁴⁴⁹⁵ - Fixing 1493
- PR #1585⁴⁴⁹⁶ - Additional Check for Inspect Tool to detect Endline Whitespace
- Issue #1584⁴⁴⁹⁷ - Clean up coroutines implementation
- PR #1583⁴⁴⁹⁸ - Adding a check for end line whitespace
- PR #1582⁴⁴⁹⁹ - Attempt to fix assert firing after scheduling loop was exited
- PR #1581⁴⁵⁰⁰ - Fixed adjacentfind_binary test
- PR #1580⁴⁵⁰¹ - Prevent some of the internal cmake lists from growing indefinitely
- PR #1579⁴⁵⁰² - Removing type_size trait, replacing it with special archive type
- Issue #1578⁴⁵⁰³ - Remove demangle_helper
- PR #1577⁴⁵⁰⁴ - Get ptr problems
- Issue #1576⁴⁵⁰⁵ - Refactor async, dataflow, and future::then
- PR #1575⁴⁵⁰⁶ - Fixing tests for parallel rotate
- PR #1574⁴⁵⁰⁷ - Cleaning up schedulers
- PR #1573⁴⁵⁰⁸ - Fixing thread pool executor
- PR #1572⁴⁵⁰⁹ - Fixing number of configured localities
- PR #1571⁴⁵¹⁰ - Reimplement decay
- PR #1570⁴⁵¹¹ - Refactoring async, apply, and dataflow APIs
- PR #1569⁴⁵¹² - Changed range for mach-o library lookup
- PR #1568⁴⁵¹³ - Mark decltype support as required
- PR #1567⁴⁵¹⁴ - Removed const from algorithms
- Issue #1566⁴⁵¹⁵ - CMAKE Configuration Test Failures for clang 3.5 on debian

⁴⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1589>

⁴⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1588>

⁴⁴⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1586>

⁴⁴⁹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1585>

⁴⁴⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1584>

⁴⁴⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1583>

⁴⁴⁹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1582>

⁴⁵⁰⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1581>

⁴⁵⁰¹ <https://github.com/STELLAR-GROUP/hpx/pull/1580>

⁴⁵⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1579>

⁴⁵⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1578>

⁴⁵⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1577>

⁴⁵⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1576>

⁴⁵⁰⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1575>

⁴⁵⁰⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1574>

⁴⁵⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1573>

⁴⁵⁰⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1572>

⁴⁵¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1571>

⁴⁵¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1570>

⁴⁵¹² <https://github.com/STELLAR-GROUP/hpx/pull/1569>

⁴⁵¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1568>

⁴⁵¹⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1567>

⁴⁵¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1566>

- PR #1565⁴⁵¹⁶ - Dylib support
- PR #1564⁴⁵¹⁷ - Converted partitioners and some algorithms to use executors
- PR #1563⁴⁵¹⁸ - Fix several #includes for Boost.Preprocessor
- PR #1562⁴⁵¹⁹ - Adding configuration option disabling/enabling all message handlers
- PR #1561⁴⁵²⁰ - Removed all occurrences of boost::move replacing it with std::move
- Issue #1560⁴⁵²¹ - Leftover HPX_REGISTER_ACTION_DECLARATION_2
- PR #1558⁴⁵²² - Revisit async/apply SFINAE conditions
- PR #1557⁴⁵²³ - Removing type_size trait, replacing it with special archive type
- PR #1556⁴⁵²⁴ - Executor algorithms
- PR #1555⁴⁵²⁵ - Remove the necessity to specify archive flags on the receiving end
- PR #1554⁴⁵²⁶ - Removing obsolete Boost.Serialization macros
- PR #1553⁴⁵²⁷ - Properly fix HPX_DEFINE_*_ACTION macros
- PR #1552⁴⁵²⁸ - Fixed algorithms relying on copy_if implementation
- PR #1551⁴⁵²⁹ - Pxfs - Modifying FindOrangeFS.cmake based on OrangeFS 2.9.X
- Issue #1550⁴⁵³⁰ - Passing plain identifier inside HPX_DEFINE_PLAIN_ACTION_1
- PR #1549⁴⁵³¹ - Fixing intel14/libstdc++4.4
- PR #1548⁴⁵³² - Moving raw_ptr to detail namespace
- PR #1547⁴⁵³³ - Adding support for executors to future.then
- PR #1546⁴⁵³⁴ - Executor traits result types
- PR #1545⁴⁵³⁵ - Integrate executors with dataflow
- PR #1543⁴⁵³⁶ - Fix potential zero-copy for primarynamespace::bulk_service_async et.al.
- PR #1542⁴⁵³⁷ - Merging HPX0.9.10 into pxfs branch
- PR #1541⁴⁵³⁸ - Removed stale cmake tests, unused since the great cmake refactoring

⁴⁵¹⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1565>

⁴⁵¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1564>

⁴⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1563>

⁴⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1562>

⁴⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1561>

⁴⁵²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1560>

⁴⁵²² <https://github.com/STELLAR-GROUP/hpx/pull/1558>

⁴⁵²³ <https://github.com/STELLAR-GROUP/hpx/pull/1557>

⁴⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1556>

⁴⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1555>

⁴⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1554>

⁴⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1553>

⁴⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1552>

⁴⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1551>

⁴⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1550>

⁴⁵³¹ <https://github.com/STELLAR-GROUP/hpx/pull/1549>

⁴⁵³² <https://github.com/STELLAR-GROUP/hpx/pull/1548>

⁴⁵³³ <https://github.com/STELLAR-GROUP/hpx/pull/1547>

⁴⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1546>

⁴⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1545>

⁴⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1543>

⁴⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1542>

⁴⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1541>

- PR #1540⁴⁵³⁹ - Fix idle-rate on platforms without TSC
- PR #1539⁴⁵⁴⁰ - Reporting situation if zero-copy-serialization was performed by a parcel generated from a plain apply/async
- PR #1538⁴⁵⁴¹ - Changed return type of bulk executors and added test
- Issue #1537⁴⁵⁴² - Incorrect cpuid config tests
- PR #1536⁴⁵⁴³ - Changed return type of bulk executors and added test
- PR #1535⁴⁵⁴⁴ - Make sure promise::get_gid() can be called more than once
- PR #1534⁴⁵⁴⁵ - Fixed async_callback with bound callback
- PR #1533⁴⁵⁴⁶ - Updated the link in the documentation to a publically- accessible URL
- PR #1532⁴⁵⁴⁷ - Make sure sync primitives are not copyable nor movable
- PR #1531⁴⁵⁴⁸ - Fix unwrapped issue with future ranges of void type
- PR #1530⁴⁵⁴⁹ - Serialization complex
- Issue #1528⁴⁵⁵⁰ - Unwrapped issue with future<void>
- Issue #1527⁴⁵⁵¹ - HPX does not build with Boost 1.58.0
- PR #1526⁴⁵⁵² - Added support for boost.multi_array serialization
- PR #1525⁴⁵⁵³ - Properly handle deferred futures, fixes #1506
- PR #1524⁴⁵⁵⁴ - Making sure invalid action argument types generate clear error message
- Issue #1522⁴⁵⁵⁵ - Need serialization support for boost multi array
- Issue #1521⁴⁵⁵⁶ - Remote async and zero-copy serialization optimizations don't play well together
- PR #1520⁴⁵⁵⁷ - Fixing UB whil registering polymorphic classes for serialization
- PR #1519⁴⁵⁵⁸ - Making detail::condition_variable safe to use
- PR #1518⁴⁵⁵⁹ - Fix when_some bug missing indices in its result
- Issue #1517⁴⁵⁶⁰ - Typo may affect CMake build system tests
- PR #1516⁴⁵⁶¹ - Fixing Posix context

⁴⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1540>

⁴⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1539>

⁴⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/pull/1538>

⁴⁵⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1537>

⁴⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/pull/1536>

⁴⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1535>

⁴⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1534>

⁴⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1533>

⁴⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1532>

⁴⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1531>

⁴⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1530>

⁴⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1528>

⁴⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1527>

⁴⁵⁵² <https://github.com/STELLAR-GROUP/hpx/pull/1526>

⁴⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/pull/1525>

⁴⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1524>

⁴⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1522>

⁴⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1521>

⁴⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1520>

⁴⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1519>

⁴⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1518>

⁴⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1517>

⁴⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/pull/1516>

- PR #1515⁴⁵⁶² - Fixing Posix context
- PR #1514⁴⁵⁶³ - Correct problems with loading dynamic components
- PR #1513⁴⁵⁶⁴ - Fixing intel glibc4 4
- Issue #1508⁴⁵⁶⁵ - memory and papi counters do not work
- Issue #1507⁴⁵⁶⁶ - Unrecognized Command Line Option Error causing exit status 0
- Issue #1506⁴⁵⁶⁷ - Properly handle deferred futures
- PR #1505⁴⁵⁶⁸ - Adding #include - would not compile without this
- Issue #1502⁴⁵⁶⁹ - boost::filesystem::exists throws unexpected exception
- Issue #1501⁴⁵⁷⁰ - hwloc configuration options are wrong for MIC
- PR #1504⁴⁵⁷¹ - Making sure boost::filesystem::exists() does not throw
- PR #1500⁴⁵⁷² - Exit application on --hpx:version/-v and --hpx:info
- PR #1498⁴⁵⁷³ - Extended task block
- PR #1497⁴⁵⁷⁴ - Unique ptr serialization
- PR #1496⁴⁵⁷⁵ - Unique ptr serialization (closed)
- PR #1495⁴⁵⁷⁶ - Switching circlegi build type to debug
- Issue #1494⁴⁵⁷⁷ - --hpx:version/-v does not exit after printing version information
- Issue #1493⁴⁵⁷⁸ - add an hpx_ prefix to libraries and components to avoid name conflicts
- Issue #1492⁴⁵⁷⁹ - Define and ensure limitations for arguments to async/apply
- PR #1489⁴⁵⁸⁰ - Enable idle rate counter on demand
- PR #1488⁴⁵⁸¹ - Made sure detail::condition_variable can be safely destroyed
- PR #1487⁴⁵⁸² - Introduced default (main) template implementation for ignore_while_checking
- PR #1486⁴⁵⁸³ - Add HPX inspect tool
- Issue #1485⁴⁵⁸⁴ - ignore_while_locked doesn't support all Lockable types

⁴⁵⁶² <https://github.com/STELLAR-GROUP/hpx/pull/1515>

⁴⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/pull/1514>

⁴⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1513>

⁴⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1508>

⁴⁵⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1507>

⁴⁵⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1506>

⁴⁵⁶⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1505>

⁴⁵⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1502>

⁴⁵⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1501>

⁴⁵⁷¹ <https://github.com/STELLAR-GROUP/hpx/pull/1504>

⁴⁵⁷² <https://github.com/STELLAR-GROUP/hpx/pull/1500>

⁴⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/pull/1498>

⁴⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1497>

⁴⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1496>

4576 <https://github.com/STELLAR-GROUP/hpx/pull/1495>4577 <https://github.com/STELLAR-GROUP/hpx/issues/1494>4578 <https://github.com/STELLAR-GROUP/hpx/issues/1493>4579 <https://github.com/STELLAR-GROUP/hpx/issues/1492>4580 <https://github.com/STELLAR-GROUP/hpx/pull/1489>4581 <https://github.com/STELLAR-GROUP/hpx/pull/1488>4582 <https://github.com/STELLAR-GROUP/hpx/pull/1487>4583 <https://github.com/STELLAR-GROUP/hpx/pull/1486>4584 <https://github.com/STELLAR-GROUP/hpx/issues/1485>

- PR #1484⁴⁵⁸⁵ - Docker image generation
- PR #1483⁴⁵⁸⁶ - Move external endian library into HPX
- PR #1482⁴⁵⁸⁷ - Actions with integer type ids
- Issue #1481⁴⁵⁸⁸ - Sync primitives safe destruction
- Issue #1480⁴⁵⁸⁹ - Move external/boost/endian into hpx/util
- Issue #1478⁴⁵⁹⁰ - Boost inspect violations
- PR #1479⁴⁵⁹¹ - Adds serialization for arrays; some further/minor fixes
- PR #1477⁴⁵⁹² - Fixing problems with the Intel compiler using a GCC 4.4 std library
- PR #1476⁴⁵⁹³ - Adding hpx::lcos::latch and hpx::lcos::local::latch
- Issue #1475⁴⁵⁹⁴ - Boost inspect violations
- PR #1473⁴⁵⁹⁵ - Fixing action move tests
- Issue #1471⁴⁵⁹⁶ - Sync primitives should not be movable
- PR #1470⁴⁵⁹⁷ - Removing hpx::util::polymorphic_factory
- PR #1468⁴⁵⁹⁸ - Fixed container creation
- Issue #1467⁴⁵⁹⁹ - HPX application fail during finalization
- Issue #1466⁴⁶⁰⁰ - HPX doesn't pick up Torque's nodefile on SuperMIC
- Issue #1464⁴⁶⁰¹ - HPX option for pre and post bootstrap performance counters
- PR #1463⁴⁶⁰² - Replacing async_colocated(id, ...) with async(colocated(id), ...)
- PR #1462⁴⁶⁰³ - Consolidated task_region with N4411
- PR #1461⁴⁶⁰⁴ - Consolidate inconsistent CMake option names
- Issue #1460⁴⁶⁰⁵ - Which malloc is actually used? or at least which one is HPX built with
- Issue #1459⁴⁶⁰⁶ - Make cmake configure step fail explicitly if compiler version is not supported
- Issue #1458⁴⁶⁰⁷ - Update parallel::task_region with N4411

⁴⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1484>

⁴⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1483>

⁴⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1482>

⁴⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1481>

⁴⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1480>

⁴⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1478>

⁴⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1479>

⁴⁵⁹² <https://github.com/STELLAR-GROUP/hpx/pull/1477>

⁴⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/pull/1476>

⁴⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1475>

⁴⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1473>

⁴⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1471>

⁴⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1470>

⁴⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1468>

⁴⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1467>

⁴⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1466>

⁴⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1464>

⁴⁶⁰² <https://github.com/STELLAR-GROUP/hpx/pull/1463>

⁴⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/pull/1462>

⁴⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1461>

⁴⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1460>

⁴⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1459>

⁴⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1458>

- PR #1456⁴⁶⁰⁸ - Consolidating new_<>()
- Issue #1455⁴⁶⁰⁹ - Replace async_colocated(id, ...) with async(colocated(id), ...)
- PR #1454⁴⁶¹⁰ - Removed harmful std::moves from return statements
- PR #1453⁴⁶¹¹ - Use range-based for-loop instead of Boost.Foreach
- PR #1452⁴⁶¹² - C++ feature tests
- PR #1451⁴⁶¹³ - When serializing, pass archive flags to traits::get_type_size
- Issue #1450⁴⁶¹⁴ - traits::get_type_size needs archive flags to enable zero_copy optimizations
- Issue #1449⁴⁶¹⁵ - “couldn’t create performance counter” - AGAS
- Issue #1448⁴⁶¹⁶ - Replace distributing factories with new_<T[]>(...)
- PR #1447⁴⁶¹⁷ - Removing obsolete remote_object component
- PR #1446⁴⁶¹⁸ - Hpx serialization
- PR #1445⁴⁶¹⁹ - Replacing travis with circleci
- PR #1443⁴⁶²⁰ - Always stripping HPX command line arguments before executing start function
- PR #1442⁴⁶²¹ - Adding --hpx:bind=none to disable thread affinities
- Issue #1439⁴⁶²² - Libraries get linked in multiple times, RPATH is not properly set
- PR #1438⁴⁶²³ - Removed superfluous typedefs
- Issue #1437⁴⁶²⁴ - hpx::init() should strip HPX-related flags from argv
- Issue #1436⁴⁶²⁵ - Add strong scaling option to htts
- PR #1435⁴⁶²⁶ - Adding async_cb, async_continue_cb, and async_colocated_cb
- PR #1434⁴⁶²⁷ - Added missing install rule, removed some dead CMake code
- PR #1433⁴⁶²⁸ - Add GitExternal and SubProject cmake scripts from eyescale/cmake repo
- Issue #1432⁴⁶²⁹ - Add command line flag to disable thread pinning
- PR #1431⁴⁶³⁰ - Fix #1423

⁴⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1456>

⁴⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1455>

⁴⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1454>

⁴⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/pull/1453>

⁴⁶¹² <https://github.com/STELLAR-GROUP/hpx/pull/1452>

⁴⁶¹³ <https://github.com/STELLAR-GROUP/hpx/pull/1451>

⁴⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1450>

⁴⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1449>

⁴⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1448>

⁴⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1447>

⁴⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1446>

⁴⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1445>

⁴⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1443>

⁴⁶²¹ <https://github.com/STELLAR-GROUP/hpx/pull/1442>

⁴⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/1439>

⁴⁶²³ <https://github.com/STELLAR-GROUP/hpx/pull/1438>

⁴⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1437>

⁴⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1436>

⁴⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1435>

⁴⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/pull/1434>

⁴⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/pull/1433>

⁴⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1432>

⁴⁶³⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1431>

- Issue #1430⁴⁶³¹ - Inconsistent CMake option names
- Issue #1429⁴⁶³² - Configure setting HPX_HAVE_PARCELPORT_MPI is ignored
- PR #1428⁴⁶³³ - Fixes #1419 (closed)
- PR #1427⁴⁶³⁴ - Adding stencil_iterator and transform_iterator
- PR #1426⁴⁶³⁵ - Fixes #1419
- PR #1425⁴⁶³⁶ - During serialization memory allocation should honour allocator chunk size
- Issue #1424⁴⁶³⁷ - chunk allocation during serialization does not use memory pool/allocator chunk size
- Issue #1423⁴⁶³⁸ - Remove HPX_STD_UNIQUE_PTR
- Issue #1422⁴⁶³⁹ - hpx:threads=all allocates too many os threads
- PR #1420⁴⁶⁴⁰ - added .travis.yml
- Issue #1419⁴⁶⁴¹ - Unify enums: hpx::runtime::state and hpx::state
- PR #1416⁴⁶⁴² - Adding travis builder
- Issue #1414⁴⁶⁴³ - Correct directory for dispatch_gcc46.hpp iteration
- Issue #1410⁴⁶⁴⁴ - Set operation algorithms
- Issue #1389⁴⁶⁴⁵ - Parallel algorithms relying on scan partitioner break for small number of elements
- Issue #1325⁴⁶⁴⁶ - Exceptions thrown during parcel handling are not handled correctly
- Issue #1315⁴⁶⁴⁷ - Errors while running performance tests
- Issue #1309⁴⁶⁴⁸ - hpx::vector partitions are not easily extendable by applications
- PR #1300⁴⁶⁴⁹ - Added serialization/de-serialization to examples.tuplespace
- Issue #1251⁴⁶⁵⁰ - hpx::threads::get_thread_count doesn't consider pending threads
- Issue #1008⁴⁶⁵¹ - Decrease in application performance overtime; occasional spikes of major slowdown
- Issue #1001⁴⁶⁵² - Zero copy serialization raises assert
- Issue #721⁴⁶⁵³ - Make HPX usable for Xeon Phi

⁴⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1430>

⁴⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/1429>

⁴⁶³³ <https://github.com/STELLAR-GROUP/hpx/pull/1428>

⁴⁶³⁴ <https://github.com/STELLAR-GROUP/hpx/pull/1427>

⁴⁶³⁵ <https://github.com/STELLAR-GROUP/hpx/pull/1426>

⁴⁶³⁶ <https://github.com/STELLAR-GROUP/hpx/pull/1425>

⁴⁶³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1424>

⁴⁶³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1423>

⁴⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1422>

⁴⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/pull/1420>

⁴⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1419>

⁴⁶⁴² <https://github.com/STELLAR-GROUP/hpx/pull/1416>

⁴⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1414>

⁴⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1410>

⁴⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1389>

⁴⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1325>

⁴⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1315>

⁴⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1309>

⁴⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/pull/1300>

⁴⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1251>

⁴⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1008>

⁴⁶⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1001>

⁴⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/721>

- Issue #524⁴⁶⁵⁴ - Extend scheduler to support threads which can't be stolen

2.10.18 HPX V0.9.10 (Mar 24, 2015)

General changes

This is the 12th official release of *HPX*. It coincides with the 7th anniversary of the first commit to our source code repository. Since then, we have seen over 12300 commits amounting to more than 220000 lines of C++ code.

The major focus of this release was to improve the reliability of large scale runs. We believe to have achieved this goal as we now can reliably run *HPX* applications on up to ~24k cores. We have also shown that *HPX* can be used with success for symmetric runs (applications using both, host cores and Intel Xeon/Phi coprocessors). This is a huge step forward in terms of the usability of *HPX*. The main focus of this work involved isolating the causes of the segmentation faults at start up and shut down. Many of these issues were discovered to be the result of the suspension of threads which hold locks.

A very important improvement introduced with this release is the refactoring of the code representing our parcel-port implementation. Parcel- ports can now be implemented by 3rd parties as independent plugins which are dynamically loaded at runtime (static linking of parcel-ports is also supported). This refactoring also includes a massive improvement of the performance of our existing parcel-ports. We were able to significantly reduce the networking latencies and to improve the available networking bandwidth. Please note that in this release we disabled the ibverbs and ipc parcel ports as those have not been ported to the new plugin system yet (see Issue #839⁴⁶⁵⁵).

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 #1338⁴⁶⁵⁶ 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`.

⁴⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/524>

⁴⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/839>

⁴⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1338>

- 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 #1402⁴⁶⁵⁷ - Internal shared_future serialization copies
- Issue #1399⁴⁶⁵⁸ - Build takes unusually long time...
- Issue #1398⁴⁶⁵⁹ - Tests using the scan partitioner are broken on at least gcc 4.7 and intel compiler
- Issue #1397⁴⁶⁶⁰ - Completely remove `hpx::unique_future`
- Issue #1396⁴⁶⁶¹ - Parallel scan algorithms with different initial values
- Issue #1395⁴⁶⁶² - Race Condition - `1d_stencil_8` - SuperMIC
- Issue #1394⁴⁶⁶³ - “suspending thread while at least one lock is being held” - `1d_stencil_8` - SuperMIC
- Issue #1393⁴⁶⁶⁴ - SEGFAULT in `1d_stencil_8` on SuperMIC
- Issue #1392⁴⁶⁶⁵ - Fixing #1168
- Issue #1391⁴⁶⁶⁶ - Parallel Algorithms for scan partitioner for small number of elements
- Issue #1387⁴⁶⁶⁷ - Failure with more than 4 localities
- Issue #1386⁴⁶⁶⁸ - Dispatching unhandled exceptions to outer user code
- Issue #1385⁴⁶⁶⁹ - Adding Copy algorithms, fixing `parallel::copy_if`
- Issue #1384⁴⁶⁷⁰ - Fixing 1325
- Issue #1383⁴⁶⁷¹ - Fixed #504: Refactor Dataflow LCO to work with futures, this removes the dataflow component as it is obsolete
- Issue #1382⁴⁶⁷² - `is_sorted`, `is_sorted_until` and `is_partitioned` algorithms
- Issue #1381⁴⁶⁷³ - fix for CMake versions prior to 3.1
- Issue #1380⁴⁶⁷⁴ - resolved warning in CMake 3.1 and newer

⁴⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1402>

⁴⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1399>

⁴⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1398>

⁴⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1397>

⁴⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1396>

⁴⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1395>

⁴⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1394>

⁴⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1393>

⁴⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1392>

⁴⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1391>

⁴⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1387>

⁴⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1386>

⁴⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1385>

⁴⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1384>

⁴⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1383>

⁴⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1382>

⁴⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1381>

⁴⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1380>

- Issue #1379⁴⁶⁷⁵ - Compilation error with papi
- Issue #1378⁴⁶⁷⁶ - Towards safer migration
- Issue #1377⁴⁶⁷⁷ - HPXConfig.cmake should include TCMALLOC_LIBRARY and TCMALLOC_INCLUDE_DIR
- Issue #1376⁴⁶⁷⁸ - Warning on uninitialized member
- Issue #1375⁴⁶⁷⁹ - Fixing 1163
- Issue #1374⁴⁶⁸⁰ - Fixing the MSVC 12 release builder
- Issue #1373⁴⁶⁸¹ - Modifying parallel search algorithm for zero length searches
- Issue #1372⁴⁶⁸² - Modifying parallel search algorithm for zero length searches
- Issue #1371⁴⁶⁸³ - Avoid holding a lock during agas::inref while doing a credit split
- Issue #1370⁴⁶⁸⁴ - --hpx:bind throws unexpected error
- Issue #1369⁴⁶⁸⁵ - Getting rid of (void) in loops
- Issue #1368⁴⁶⁸⁶ - Variadic templates support for tuple
- Issue #1367⁴⁶⁸⁷ - One last batch of variadic templates support
- Issue #1366⁴⁶⁸⁸ - Fixing symbolic namespace hang
- Issue #1365⁴⁶⁸⁹ - More held locks
- Issue #1364⁴⁶⁹⁰ - Add counters 1363
- Issue #1363⁴⁶⁹¹ - Add thread overhead counters
- Issue #1362⁴⁶⁹² - Std config removal
- Issue #1361⁴⁶⁹³ - Parcelport plugins
- Issue #1360⁴⁶⁹⁴ - Detuplify transfer_action
- Issue #1359⁴⁶⁹⁵ - Removed obsolete checks
- Issue #1358⁴⁶⁹⁶ - Fixing 1352
- Issue #1357⁴⁶⁹⁷ - Variadic templates support for runtime_support and components

⁴⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1379>

⁴⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1378>

⁴⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1377>

⁴⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1376>

⁴⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1375>

⁴⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1374>

⁴⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1373>

⁴⁶⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1372>

⁴⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1371>

⁴⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1370>

⁴⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1369>

⁴⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1368>

⁴⁶⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1367>

⁴⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1366>

⁴⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1365>

⁴⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1364>

⁴⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1363>

⁴⁶⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1362>

⁴⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1361>

⁴⁶⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1360>

⁴⁶⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1359>

⁴⁶⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1358>

⁴⁶⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1357>

- Issue #1356⁴⁶⁹⁸ - fixed coordinate test for intel13
- Issue #1355⁴⁶⁹⁹ - fixed coordinate.hpp
- Issue #1354⁴⁷⁰⁰ - Lexicographical Compare completed
- Issue #1353⁴⁷⁰¹ - HPX should set Boost_ADDITIONAL_VERSIONS flags
- Issue #1352⁴⁷⁰² - Error: Cannot find action ‘‘ in type registry: HPX(bad_action_code)
- Issue #1351⁴⁷⁰³ - Variadic templates support for applicers
- Issue #1350⁴⁷⁰⁴ - Actions simplification
- Issue #1349⁴⁷⁰⁵ - Variadic when and wait functions
- Issue #1348⁴⁷⁰⁶ - Added hpx_init header to test files
- Issue #1347⁴⁷⁰⁷ - Another batch of variadic templates support
- Issue #1346⁴⁷⁰⁸ - Segmented copy
- Issue #1345⁴⁷⁰⁹ - Attempting to fix hangs during shutdown
- Issue #1344⁴⁷¹⁰ - Std config removal
- Issue #1343⁴⁷¹¹ - Removing various distribution policies for hpx::vector
- Issue #1342⁴⁷¹² - Inclusive scan
- Issue #1341⁴⁷¹³ - Exclusive scan
- Issue #1340⁴⁷¹⁴ - Adding parallel::count for distributed data structures, adding tests
- Issue #1339⁴⁷¹⁵ - Update argument order for transform_reduce
- Issue #1337⁴⁷¹⁶ - Fix dataflow to handle properly ranges of futures
- Issue #1336⁴⁷¹⁷ - dataflow needs to hold onto futures passed to it
- Issue #1335⁴⁷¹⁸ - Fails to compile with msvc14
- Issue #1334⁴⁷¹⁹ - Examples build problem
- Issue #1333⁴⁷²⁰ - Distributed transform reduce

⁴⁶⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1356>

⁴⁶⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1355>

⁴⁷⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1354>

⁴⁷⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1353>

⁴⁷⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1352>

⁴⁷⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1351>

⁴⁷⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1350>

⁴⁷⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1349>

⁴⁷⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1348>

⁴⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1347>

⁴⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1346>

⁴⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1345>

⁴⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1344>

⁴⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1343>

⁴⁷¹² <https://github.com/STELLAR-GROUP/hpx/issues/1342>

⁴⁷¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1341>

⁴⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1340>

⁴⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1339>

⁴⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1337>

⁴⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1336>

⁴⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1335>

⁴⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1334>

⁴⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1333>

- Issue #1332⁴⁷²¹ - Variadic templates support for actions
- Issue #1331⁴⁷²² - Some ambiguous calls of map::erase have been prevented by adding additional check in locality constructor.
- Issue #1330⁴⁷²³ - Defining Plain Actions does not work as described in the documentation
- Issue #1329⁴⁷²⁴ - Distributed vector cleanup
- Issue #1328⁴⁷²⁵ - Sync docs and comments with code in hello_world example
- Issue #1327⁴⁷²⁶ - Typos in docs
- Issue #1326⁴⁷²⁷ - Documentation and code diverged in Fibonacci tutorial
- Issue #1325⁴⁷²⁸ - Exceptions thrown during parcel handling are not handled correctly
- Issue #1324⁴⁷²⁹ - fixed bandwidth calculation
- Issue #1323⁴⁷³⁰ - mmap() failed to allocate thread stack due to insufficient resources
- Issue #1322⁴⁷³¹ - HPX fails to build aa182cf
- Issue #1321⁴⁷³² - Limiting size of outgoing messages while coalescing parcels
- Issue #1320⁴⁷³³ - passing a future with launch::deferred in remote function call causes hang
- Issue #1319⁴⁷³⁴ - An exception when tries to specify number high priority threads with abp-priority
- Issue #1318⁴⁷³⁵ - Unable to run program with abp-priority and numa-sensitivity enabled
- Issue #1317⁴⁷³⁶ - N4071 Search/Search_n finished, minor changes
- Issue #1316⁴⁷³⁷ - Add config option to make -Ihpix.run_hpx_main!=1 the default
- Issue #1314⁴⁷³⁸ - Variadic support for async and apply
- Issue #1313⁴⁷³⁹ - Adjust when_any/some to the latest proposed interfaces
- Issue #1312⁴⁷⁴⁰ - Fixing #857: hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #1311⁴⁷⁴¹ - Distributed get'er/set'er_values for distributed vector
- Issue #1310⁴⁷⁴² - Crashing in hpx::parcelset::policies::mpi::connection_handler::handle_messages() on Super-MIC

⁴⁷²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1332>

⁴⁷²² <https://github.com/STELLAR-GROUP/hpx/issues/1331>

⁴⁷²³ <https://github.com/STELLAR-GROUP/hpx/issues/1330>

⁴⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1329>

⁴⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1328>

⁴⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1327>

⁴⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1326>

⁴⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1325>

⁴⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1324>

⁴⁷³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1323>

⁴⁷³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1322>

⁴⁷³² <https://github.com/STELLAR-GROUP/hpx/issues/1321>

⁴⁷³³ <https://github.com/STELLAR-GROUP/hpx/issues/1320>

⁴⁷³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1319>

⁴⁷³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1318>

⁴⁷³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1317>

⁴⁷³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1316>

⁴⁷³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1314>

⁴⁷³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1313>

⁴⁷⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1312>

⁴⁷⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1311>

⁴⁷⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1310>

- Issue #1308⁴⁷⁴³ - Unable to execute an application with –hpx:threads
- Issue #1307⁴⁷⁴⁴ - merge_graph linking issue
- Issue #1306⁴⁷⁴⁵ - First batch of variadic templates support
- Issue #1305⁴⁷⁴⁶ - Create a compiler wrapper
- Issue #1304⁴⁷⁴⁷ - Provide a compiler wrapper for hpx
- Issue #1303⁴⁷⁴⁸ - Drop support for GCC44
- Issue #1302⁴⁷⁴⁹ - Fixing #1297
- Issue #1301⁴⁷⁵⁰ - Compilation error when tried to use boost range iterators with wait_all
- Issue #1298⁴⁷⁵¹ - Distributed vector
- Issue #1297⁴⁷⁵² - Unable to invoke component actions recursively
- Issue #1294⁴⁷⁵³ - HDF5 build error
- Issue #1275⁴⁷⁵⁴ - The parcelport implementation is non-optimal
- Issue #1267⁴⁷⁵⁵ - Added classes and unit tests for local_file, orangefs_file and pufs_file
- Issue #1264⁴⁷⁵⁶ - Error “assertion ‘!m_fun’ failed” randomly occurs when using TCP
- Issue #1254⁴⁷⁵⁷ - thread binding seems to not work properly
- Issue #1220⁴⁷⁵⁸ - parallel::copy_if is broken
- Issue #1217⁴⁷⁵⁹ - Find a better way of fixing the issue patched by #1216
- Issue #1168⁴⁷⁶⁰ - Starting HPX on Cray machines using aprun isn’t working correctly
- Issue #1085⁴⁷⁶¹ - Replace startup and shutdown barriers with broadcasts
- Issue #981⁴⁷⁶² - With SLURM, –hpx:threads=8 should not be necessary
- Issue #857⁴⁷⁶³ - hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #850⁴⁷⁶⁴ - “flush” not documented
- Issue #763⁴⁷⁶⁵ - Create buildbot instance that uses std::bind as HPX_STD_BIND

⁴⁷⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1308>

⁴⁷⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1307>

⁴⁷⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1306>

⁴⁷⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1305>

⁴⁷⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1304>

⁴⁷⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1303>

⁴⁷⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1302>

⁴⁷⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1301>

⁴⁷⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1298>

⁴⁷⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1297>

⁴⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1294>

⁴⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1275>

⁴⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1267>

⁴⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1264>

⁴⁷⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1254>

⁴⁷⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1220>

⁴⁷⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1217>

⁴⁷⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1168>

⁴⁷⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1085>

⁴⁷⁶² <https://github.com/STELLAR-GROUP/hpx/issues/981>

⁴⁷⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/857>

⁴⁷⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/850>

⁴⁷⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/763>

- Issue #680⁴⁷⁶⁶ - Convert parcel ports into a plugin system
- Issue #582⁴⁷⁶⁷ - Make exception thrown from HPX threads available from `hpx::init`
- Issue #504⁴⁷⁶⁸ - Refactor Dataflow LCO to work with futures
- Issue #196⁴⁷⁶⁹ - Don't store copies of the locality network metadata in the gva table

2.10.19 HPX V0.9.9 (Oct 31, 2014, codename Spooky)

General changes

We have had over 1500 commits since the last release and we have closed over 200 tickets (bugs, feature requests, pull requests, etc.). These are by far the largest numbers of commits and resolved issues for any of the *HPX* releases so far. We are especially happy about the large number of people who contributed for the first time to *HPX*.

- We completed the transition from the older (non-conforming) implementation of `hpx::future` to the new and fully conforming version by removing the old code and by renaming the type `hpx::unique_future` to `hpx::future`. In order to maintain backwards compatibility with existing code which uses the type `hpx::unique_future` we support the configuration variable `HPX_UNIQUE_FUTURE_ALIAS`. If this variable is set to ON while running cmake it will additionally define a template alias for this type.
- We rewrote and significantly changed our build system. Please have a look at the new (now generated) documentation here: [Building HPX](#). 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 #1204⁴⁷⁷⁰.
- We started to implement various proposals to the C++ Standardization committee related to parallelism and concurrency, most notably N4409⁴⁷⁷¹ (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411⁴⁷⁷² (Task Region Rev. 3), and N4313⁴⁷⁷³ (Working Draft, Technical Specification for C++ Extensions for Concurrency).
- We completely remodeled our automatic build system to run builds and unit tests on various systems and compilers. This allows us to find most bugs right as they were introduced and helps to maintain a high level of quality and compatibility. The newest build logs can be found at [HPX Buildbot Website](#)⁴⁷⁷⁴.

⁴⁷⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/680>

⁴⁷⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/582>

⁴⁷⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/504>

⁴⁷⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/196>

⁴⁷⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1204>

⁴⁷⁷¹ <http://wg21.link/n4409>

⁴⁷⁷² <http://wg21.link/n4411>

⁴⁷⁷³ <http://wg21.link/n4313>

⁴⁷⁷⁴ <http://rostam.cct.lsu.edu/>

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1296⁴⁷⁷⁵ - Rename make_error_future to make_exceptional_future, adjust to N4123
- Issue #1295⁴⁷⁷⁶ - building issue
- Issue #1293⁴⁷⁷⁷ - Transpose example
- Issue #1292⁴⁷⁷⁸ - Wrong abs() function used in example
- Issue #1291⁴⁷⁷⁹ - non-synchronized shift operators have been removed
- Issue #1290⁴⁷⁸⁰ - RDTSCP is defined as true for Xeon Phi build
- Issue #1289⁴⁷⁸¹ - Fixing 1288
- Issue #1288⁴⁷⁸² - Add new performance counters
- Issue #1287⁴⁷⁸³ - Hierarchy scheduler broken performance counters
- Issue #1286⁴⁷⁸⁴ - Algorithm cleanup
- Issue #1285⁴⁷⁸⁵ - Broken Links in Documentation
- Issue #1284⁴⁷⁸⁶ - Uninitialized copy
- Issue #1283⁴⁷⁸⁷ - missing boost::scoped_ptr includes
- Issue #1282⁴⁷⁸⁸ - Update documentation of build options for schedulers
- Issue #1281⁴⁷⁸⁹ - reset idle rate counter
- Issue #1280⁴⁷⁹⁰ - Bug when executing on Intel MIC
- Issue #1279⁴⁷⁹¹ - Add improved when_all/wait_all
- Issue #1278⁴⁷⁹² - Implement improved when_all/wait_all
- Issue #1277⁴⁷⁹³ - feature request: get access to argc argv and variables_map
- Issue #1276⁴⁷⁹⁴ - Remove merging map
- Issue #1274⁴⁷⁹⁵ - Weird (wrong) string code in papi.cpp

⁴⁷⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1296>

⁴⁷⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1295>

⁴⁷⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1293>

⁴⁷⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1292>

⁴⁷⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1291>

⁴⁷⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1290>

⁴⁷⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1289>

⁴⁷⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1288>

⁴⁷⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1287>

⁴⁷⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1286>

⁴⁷⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1285>

⁴⁷⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1284>

⁴⁷⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1283>

⁴⁷⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1282>

⁴⁷⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1281>

⁴⁷⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1280>

⁴⁷⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1279>

⁴⁷⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1278>

⁴⁷⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1277>

⁴⁷⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1276>

⁴⁷⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1274>

- Issue #1273⁴⁷⁹⁶ - Sequential task execution policy
- Issue #1272⁴⁷⁹⁷ - Avoid CMake name clash for Boost.Thread library
- Issue #1271⁴⁷⁹⁸ - Updates on HPX Test Units
- Issue #1270⁴⁷⁹⁹ - hpx/util/safe_lexical_cast.hpp is added
- Issue #1269⁴⁸⁰⁰ - Added default value for “LIB” cmake variable
- Issue #1268⁴⁸⁰¹ - Memory Counters not working
- Issue #1266⁴⁸⁰² - FindHPX.cmake is not installed
- Issue #1263⁴⁸⁰³ - apply_remote test takes too long
- Issue #1262⁴⁸⁰⁴ - Chrono cleanup
- Issue #1261⁴⁸⁰⁵ - Need make install for papi counters and this builds all the examples
- Issue #1260⁴⁸⁰⁶ - Documentation of Stencil example claims
- Issue #1259⁴⁸⁰⁷ - Avoid double-linking Boost on Windows
- Issue #1257⁴⁸⁰⁸ - Adding additional parameter to create_thread
- Issue #1256⁴⁸⁰⁹ - added buildbot changes to release notes
- Issue #1255⁴⁸¹⁰ - Cannot build MiniGhost
- Issue #1253⁴⁸¹¹ - hpx::thread defects
- Issue #1252⁴⁸¹² - HPX_PREFIX is too fragile
- Issue #1250⁴⁸¹³ - switch_to_fiber_emulation does not work properly
- Issue #1249⁴⁸¹⁴ - Documentation is generated under Release folder
- Issue #1248⁴⁸¹⁵ - Fix usage of hpx_generic_coroutine_context and get tests passing on powerpc
- Issue #1247⁴⁸¹⁶ - Dynamic linking error
- Issue #1246⁴⁸¹⁷ - Make cpuid.cpp C++11 compliant
- Issue #1245⁴⁸¹⁸ - HPX fails on startup (setting thread affinity mask)

⁴⁷⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1273>

⁴⁷⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1272>

⁴⁷⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1271>

⁴⁷⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1270>

⁴⁸⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1269>

⁴⁸⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/1268>

⁴⁸⁰² <https://github.com/STELLAR-GROUP/hpx/issues/1266>

⁴⁸⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1263>

⁴⁸⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1262>

⁴⁸⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1261>

⁴⁸⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1260>

⁴⁸⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1259>

⁴⁸⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1257>

⁴⁸⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1256>

⁴⁸¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1255>

⁴⁸¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1253>

⁴⁸¹² <https://github.com/STELLAR-GROUP/hpx/issues/1252>

⁴⁸¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1250>

⁴⁸¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1249>

⁴⁸¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1248>

⁴⁸¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1247>

⁴⁸¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1246>

⁴⁸¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1245>

- Issue #1244⁴⁸¹⁹ - HPX_WITH_RDTSC configure test fails, but should succeed
- Issue #1243⁴⁸²⁰ - CTest dashboard info for CSCS CDash drop location
- Issue #1242⁴⁸²¹ - Mac fixes
- Issue #1241⁴⁸²² - Failure in Distributed with Boost 1.56
- Issue #1240⁴⁸²³ - fix a race condition in examples.diskperf
- Issue #1239⁴⁸²⁴ - fix wait_each in examples.diskperf
- Issue #1238⁴⁸²⁵ - Fixed #1237: hpx::util::portable_binary_iarchive failed
- Issue #1237⁴⁸²⁶ - hpx::util::portable_binary_iarchive faileds
- Issue #1235⁴⁸²⁷ - Fixing clang warnings and errors
- Issue #1234⁴⁸²⁸ - TCP runs fail: Transport endpoint is not connected
- Issue #1233⁴⁸²⁹ - Making sure the correct number of threads is registered with AGAS
- Issue #1232⁴⁸³⁰ - Fixing race in wait_xxx
- Issue #1231⁴⁸³¹ - Parallel minmax
- Issue #1230⁴⁸³² - Distributed run of 1d_stencil_8 uses less threads than spec. & sometimes gives errors
- Issue #1229⁴⁸³³ - Unstable number of threads
- Issue #1228⁴⁸³⁴ - HPX link error (cmake / MPI)
- Issue #1226⁴⁸³⁵ - Warning about struct/class thread_counters
- Issue #1225⁴⁸³⁶ - Adding parallel::replace etc
- Issue #1224⁴⁸³⁷ - Extending dataflow to pass through non-future arguments
- Issue #1223⁴⁸³⁸ - Remaining find algorithms implemented, N4071
- Issue #1222⁴⁸³⁹ - Merging all the changes
- Issue #1221⁴⁸⁴⁰ - No error output when using mpirun with hpx
- Issue #1219⁴⁸⁴¹ - Adding new AGAS cache performance counters

⁴⁸¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1244>

⁴⁸²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1243>

⁴⁸²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1242>

⁴⁸²² <https://github.com/STELLAR-GROUP/hpx/issues/1241>

⁴⁸²³ <https://github.com/STELLAR-GROUP/hpx/issues/1240>

⁴⁸²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1239>

⁴⁸²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1238>

⁴⁸²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1237>

⁴⁸²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1235>

⁴⁸²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1234>

⁴⁸²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1233>

⁴⁸³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1232>

⁴⁸³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1231>

⁴⁸³² <https://github.com/STELLAR-GROUP/hpx/issues/1230>

⁴⁸³³ <https://github.com/STELLAR-GROUP/hpx/issues/1229>

⁴⁸³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1228>

⁴⁸³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1226>

⁴⁸³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1225>

⁴⁸³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1224>

⁴⁸³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1223>

⁴⁸³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1222>

⁴⁸⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1221>

⁴⁸⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1219>

- Issue #1216⁴⁸⁴² - Fixing using futures (clients) as arguments to actions
- Issue #1215⁴⁸⁴³ - Error compiling simple component
- Issue #1214⁴⁸⁴⁴ - Stencil docs
- Issue #1213⁴⁸⁴⁵ - Using more than a few dozen MPI processes on SuperMike results in a seg fault before getting to hpx_main
- Issue #1212⁴⁸⁴⁶ - Parallel rotate
- Issue #1211⁴⁸⁴⁷ - Direct actions cause the future's shared_state to be leaked
- Issue #1210⁴⁸⁴⁸ - Refactored local::promise to be standard conformant
- Issue #1209⁴⁸⁴⁹ - Improve command line handling
- Issue #1208⁴⁸⁵⁰ - Adding parallel::reverse and parallel::reverse_copy
- Issue #1207⁴⁸⁵¹ - Add copy_backward and move_backward
- Issue #1206⁴⁸⁵² - N4071 additional algorithms implemented
- Issue #1204⁴⁸⁵³ - Cmake simplification and various other minor changes
- Issue #1203⁴⁸⁵⁴ - Implementing new launch policy for (local) async: hpx::launch::fork.
- Issue #1202⁴⁸⁵⁵ - Failed assertion in connection_cache.hpp
- Issue #1201⁴⁸⁵⁶ - pkg-config doesn't add mpi link directories
- Issue #1200⁴⁸⁵⁷ - Error when querying time performance counters
- Issue #1199⁴⁸⁵⁸ - library path is now configurable (again)
- Issue #1198⁴⁸⁵⁹ - Error when querying performance counters
- Issue #1197⁴⁸⁶⁰ - tests fail with intel compiler
- Issue #1196⁴⁸⁶¹ - Silence several warnings
- Issue #1195⁴⁸⁶² - Rephrase initializers to work with VC++ 2012
- Issue #1194⁴⁸⁶³ - Simplify parallel algorithms
- Issue #1193⁴⁸⁶⁴ - Adding parallel::equal

⁴⁸⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1216>

⁴⁸⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1215>

⁴⁸⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1214>

⁴⁸⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1213>

⁴⁸⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1212>

⁴⁸⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1211>

⁴⁸⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1210>

⁴⁸⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1209>

⁴⁸⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1208>

⁴⁸⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1207>

⁴⁸⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1206>

⁴⁸⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1204>

⁴⁸⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1203>

⁴⁸⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1202>

⁴⁸⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1201>

⁴⁸⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1200>

⁴⁸⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1199>

⁴⁸⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1198>

⁴⁸⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1197>

⁴⁸⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1196>

⁴⁸⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1195>

⁴⁸⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1194>

⁴⁸⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1193>

- Issue #1192⁴⁸⁶⁵ - HPX(out_of_memory) on including <hpx/hpx.hpp>
- Issue #1191⁴⁸⁶⁶ - Fixing #1189
- Issue #1190⁴⁸⁶⁷ - Chrono cleanup
- Issue #1189⁴⁸⁶⁸ - Deadlock .. somewhere? (probably serialization)
- Issue #1188⁴⁸⁶⁹ - Removed `future::get_status()`
- Issue #1186⁴⁸⁷⁰ - Fixed FindOpenCL to find current AMD APP SDK
- Issue #1184⁴⁸⁷¹ - Tweaking future unwrapping
- Issue #1183⁴⁸⁷² - Extended `parallel::reduce`
- Issue #1182⁴⁸⁷³ - `future::unwrap` hangs for `launch::deferred`
- Issue #1181⁴⁸⁷⁴ - Adding `all_of`, `any_of`, and `none_of` and corresponding documentation
- Issue #1180⁴⁸⁷⁵ - `hpx::cout` defect
- Issue #1179⁴⁸⁷⁶ - `hpx::async` does not work for member function pointers when called on types with self-defined unary `operator*`
- Issue #1178⁴⁸⁷⁷ - Implemented variadic `hpx::util::zip_iterator`
- Issue #1177⁴⁸⁷⁸ - MPI parcelport defect
- Issue #1176⁴⁸⁷⁹ - `HPX_DEFINE_COMPONENT_CONST_ACTION_TPL` does not have a 2-argument version
- Issue #1175⁴⁸⁸⁰ - Create `util::zip_iterator` working with `util::tuple<>`
- Issue #1174⁴⁸⁸¹ - Error Building HPX on linux, `root_certificate_authority.cpp`
- Issue #1173⁴⁸⁸² - `hpx::cout` output lost
- Issue #1172⁴⁸⁸³ - HPX build error with Clang 3.4.2
- Issue #1171⁴⁸⁸⁴ - `CMAKE_INSTALL_PREFIX` ignored
- Issue #1170⁴⁸⁸⁵ - Close `hpx_benchmarks` repository on Github
- Issue #1169⁴⁸⁸⁶ - Buildbot emails have syntax error in url
- Issue #1167⁴⁸⁸⁷ - Merge partial implementation of standards proposal N3960

⁴⁸⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1192>

⁴⁸⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1191>

⁴⁸⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1190>

⁴⁸⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1189>

⁴⁸⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1188>

⁴⁸⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1186>

⁴⁸⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/1184>

⁴⁸⁷² <https://github.com/STELLAR-GROUP/hpx/issues/1183>

⁴⁸⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/1182>

⁴⁸⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1181>

⁴⁸⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1180>

⁴⁸⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1179>

⁴⁸⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1178>

⁴⁸⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1177>

⁴⁸⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1176>

⁴⁸⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1175>

⁴⁸⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1174>

⁴⁸⁸² <https://github.com/STELLAR-GROUP/hpx/issues/1173>

⁴⁸⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/1172>

⁴⁸⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1171>

⁴⁸⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1170>

⁴⁸⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1169>

⁴⁸⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1167>

- Issue #1166⁴⁸⁸⁸ - Fixed several compiler warnings
- Issue #1165⁴⁸⁸⁹ - cmake warns: “tests.regressions.actions” does not exist
- Issue #1164⁴⁸⁹⁰ - Want my own serialization of hpx::future
- Issue #1162⁴⁸⁹¹ - Segfault in hello_world example
- Issue #1161⁴⁸⁹² - Use HPX_ASSERT to aid the compiler
- Issue #1160⁴⁸⁹³ - Do not put -DNDEBUG into hpx_application.pc
- Issue #1159⁴⁸⁹⁴ - Support Clang 3.4.2
- Issue #1158⁴⁸⁹⁵ - Fixed #1157: Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1157⁴⁸⁹⁶ - Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1156⁴⁸⁹⁷ - Force inlining fails
- Issue #1155⁴⁸⁹⁸ - changed header of printout to be compatible with python csv module
- Issue #1154⁴⁸⁹⁹ - Fixing iostreams
- Issue #1153⁴⁹⁰⁰ - Standard manipulators (like std::endl) do not work with hpx::ostream
- Issue #1152⁴⁹⁰¹ - Functions revamp
- Issue #1151⁴⁹⁰² - Suppressing cmake 3.0 policy warning for CMP0026
- Issue #1150⁴⁹⁰³ - Client Serialization error
- Issue #1149⁴⁹⁰⁴ - Segfault on Stampede
- Issue #1148⁴⁹⁰⁵ - Refactoring mini-ghost
- Issue #1147⁴⁹⁰⁶ - N3960 copy_if and copy_n implemented and tested
- Issue #1146⁴⁹⁰⁷ - Stencil print
- Issue #1145⁴⁹⁰⁸ - N3960 hpx::parallel::copy implemented and tested
- Issue #1144⁴⁹⁰⁹ - OpenMP examples 1d_stencil do not build
- Issue #1143⁴⁹¹⁰ - 1d_stencil OpenMP examples do not build

⁴⁸⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1166>

⁴⁸⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1165>

⁴⁸⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1164>

⁴⁸⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1162>

⁴⁸⁹² <https://github.com/STELLAR-GROUP/hpx/issues/1161>

⁴⁸⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/1160>

⁴⁸⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1159>

⁴⁸⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1158>

⁴⁸⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1157>

4897 <https://github.com/STELLAR-GROUP/hpx/issues/1156>4898 <https://github.com/STELLAR-GROUP/hpx/issues/1155>4899 <https://github.com/STELLAR-GROUP/hpx/issues/1154>4900 <https://github.com/STELLAR-GROUP/hpx/issues/1153>4901 <https://github.com/STELLAR-GROUP/hpx/issues/1152>4902 <https://github.com/STELLAR-GROUP/hpx/issues/1151>4903 <https://github.com/STELLAR-GROUP/hpx/issues/1150>4904 <https://github.com/STELLAR-GROUP/hpx/issues/1149>4905 <https://github.com/STELLAR-GROUP/hpx/issues/1148>4906 <https://github.com/STELLAR-GROUP/hpx/issues/1147>4907 <https://github.com/STELLAR-GROUP/hpx/issues/1146>4908 <https://github.com/STELLAR-GROUP/hpx/issues/1145>4909 <https://github.com/STELLAR-GROUP/hpx/issues/1144>4910 <https://github.com/STELLAR-GROUP/hpx/issues/1143>

- Issue #1142⁴⁹¹¹ - Cannot build HPX with gcc 4.6 on OS X
- Issue #1140⁴⁹¹² - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
- Issue #1139⁴⁹¹³ - hpx/hpx/config/compiler_specific.hpp
- Issue #1138⁴⁹¹⁴ - clean up pkg-config files
- Issue #1137⁴⁹¹⁵ - Improvements to create binary packages
- Issue #1136⁴⁹¹⁶ - HPX_GCC_VERSION not defined on all compilers
- Issue #1135⁴⁹¹⁷ - Avoiding collision between winsock2.h and windows.h
- Issue #1134⁴⁹¹⁸ - Making sure, that hpx::finalize can be called from any locality
- Issue #1133⁴⁹¹⁹ - 1d stencil examples
- Issue #1131⁴⁹²⁰ - Refactor unique_function implementation
- Issue #1130⁴⁹²¹ - Unique function
- Issue #1129⁴⁹²² - Some fixes to the Build system on OS X
- Issue #1128⁴⁹²³ - Action future args
- Issue #1127⁴⁹²⁴ - Executor causes segmentation fault
- Issue #1124⁴⁹²⁵ - Adding new API functions: register_id_with_basename, unregister_id_with_basename, find_ids_from_basename; adding test
- Issue #1123⁴⁹²⁶ - Reduce nesting of try-catch construct in encode_parcels?
- Issue #1122⁴⁹²⁷ - Client base fixes
- Issue #1121⁴⁹²⁸ - Update hpxrun.py.in
- Issue #1120⁴⁹²⁹ - HTTS2 tests compile errors on v110 (VS2012)
- Issue #1119⁴⁹³⁰ - Remove references to boost::atomic in accumulator example
- Issue #1118⁴⁹³¹ - Only build test thread_pool_executor_1114_test if HPX_SCHEDULER is set
- Issue #1117⁴⁹³² - local_queue_executor linker error on vc110
- Issue #1116⁴⁹³³ - Disabled performance counter should give runtime errors, not invalid data

⁴⁹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1142>

⁴⁹¹² <https://github.com/STELLAR-GROUP/hpx/issues/1140>

⁴⁹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1139>

⁴⁹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1138>

⁴⁹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1137>

⁴⁹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1136>

⁴⁹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1135>

⁴⁹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1134>

⁴⁹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1133>

⁴⁹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1131>

⁴⁹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1130>

⁴⁹²² <https://github.com/STELLAR-GROUP/hpx/issues/1129>

⁴⁹²³ <https://github.com/STELLAR-GROUP/hpx/issues/1128>

⁴⁹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1127>

⁴⁹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1124>

⁴⁹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1123>

⁴⁹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1122>

⁴⁹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1121>

⁴⁹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1120>

⁴⁹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1119>

⁴⁹³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1118>

⁴⁹³² <https://github.com/STELLAR-GROUP/hpx/issues/1117>

⁴⁹³³ <https://github.com/STELLAR-GROUP/hpx/issues/1116>

- Issue #1115⁴⁹³⁴ - Compile error with Intel C++ 13.1
- Issue #1114⁴⁹³⁵ - Default constructed executor is not usable
- Issue #1113⁴⁹³⁶ - Fast compilation of logging causes ABI incompatibilities between different NDEBUG values
- Issue #1112⁴⁹³⁷ - Using thread_pool_executors causes segfault
- Issue #1111⁴⁹³⁸ - `hpx::threads::get_thread_data` always returns zero
- Issue #1110⁴⁹³⁹ - Remove unnecessary null pointer checks
- Issue #1109⁴⁹⁴⁰ - More tests adjustments
- Issue #1108⁴⁹⁴¹ - Clarify build rules for “libboost_atomic-mt.so”?
- Issue #1107⁴⁹⁴² - Remove unnecessary null pointer checks
- Issue #1106⁴⁹⁴³ - network_storage benchmark improvements, adding legends to plots and tidying layout
- Issue #1105⁴⁹⁴⁴ - Add more plot outputs and improve instructions doc
- Issue #1104⁴⁹⁴⁵ - Complete quoting for parameters of some CMake commands
- Issue #1103⁴⁹⁴⁶ - Work on test/scripts
- Issue #1102⁴⁹⁴⁷ - Changed minimum requirement of window install to 2012
- Issue #1101⁴⁹⁴⁸ - Changed minimum requirement of window install to 2012
- Issue #1100⁴⁹⁴⁹ - Changed readme to no longer specify using MSVC 2010 compiler
- Issue #1099⁴⁹⁵⁰ - Error returning futures from component actions
- Issue #1098⁴⁹⁵¹ - Improve storage test
- Issue #1097⁴⁹⁵² - data_actions quickstart example calls missing function decorate_action of data_get_action
- Issue #1096⁴⁹⁵³ - MPI parcelport broken with new zero copy optimization
- Issue #1095⁴⁹⁵⁴ - Warning C4005: _WIN32_WINNT: Macro redefinition
- Issue #1094⁴⁹⁵⁵ - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
- Issue #1093⁴⁹⁵⁶ - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS

⁴⁹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1115>

⁴⁹³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1114>

⁴⁹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1113>

⁴⁹³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1112>

⁴⁹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1111>

⁴⁹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1110>

⁴⁹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1109>

⁴⁹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1108>

⁴⁹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1107>

⁴⁹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1106>

⁴⁹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1105>

⁴⁹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1104>

⁴⁹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1103>

⁴⁹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1102>

⁴⁹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1101>

⁴⁹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1100>

⁴⁹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1099>

⁴⁹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1098>

⁴⁹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1097>

⁴⁹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1096>

⁴⁹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1095>

⁴⁹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1094>

⁴⁹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1093>

- Issue #1092⁴⁹⁵⁷ - Rename unique_future<> back to future<>
- Issue #1091⁴⁹⁵⁸ - Inconsistent error message
- Issue #1090⁴⁹⁵⁹ - On windows 8.1 the examples crashed if using more than one os thread
- Issue #1089⁴⁹⁶⁰ - Components should be allowed to have their own executor
- Issue #1088⁴⁹⁶¹ - Add possibility to select a network interface for the ibverbs parcelport
- Issue #1087⁴⁹⁶² - ibverbs and ipc parcelport uses zero copy optimization
- Issue #1083⁴⁹⁶³ - Make shell examples copyable in docs
- Issue #1082⁴⁹⁶⁴ - Implement proper termination detection during shutdown
- Issue #1081⁴⁹⁶⁵ - Implement thread_specific_ptr for hpx::threads
- Issue #1072⁴⁹⁶⁶ - make install not working properly
- Issue #1070⁴⁹⁶⁷ - Complete quoting for parameters of some CMake commands
- Issue #1059⁴⁹⁶⁸ - Fix more unused variable warnings
- Issue #1051⁴⁹⁶⁹ - Implement when_each
- Issue #973⁴⁹⁷⁰ - Would like option to report hwloc bindings
- Issue #970⁴⁹⁷¹ - Bad flags for Fortran compiler
- Issue #941⁴⁹⁷² - Create a proper user level context switching class for BG/Q
- Issue #935⁴⁹⁷³ - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #934⁴⁹⁷⁴ - Want to build HPX without dynamic libraries
- Issue #927⁴⁹⁷⁵ - Make hpx/lcos/reduce.hpp accept futures of id_type
- Issue #926⁴⁹⁷⁶ - All unit tests that are run with more than one thread with CTest/hpx_run_test should configure hpx.os_threads
- Issue #925⁴⁹⁷⁷ - regression_dataflow_791 needs to be brought in line with HPX standards
- Issue #899⁴⁹⁷⁸ - Fix race conditions in regression tests
- Issue #879⁴⁹⁷⁹ - Hung test leads to cascading test failure; make tests should support the MPI parcelport

⁴⁹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1092>

⁴⁹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1091>

⁴⁹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1090>

⁴⁹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1089>

⁴⁹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1088>

⁴⁹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1087>

⁴⁹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1083>

⁴⁹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1082>

⁴⁹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1081>

⁴⁹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1072>

⁴⁹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1070>

⁴⁹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1059>

⁴⁹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1051>

⁴⁹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/973>

⁴⁹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/970>

⁴⁹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/941>

⁴⁹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/935>

⁴⁹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/934>

⁴⁹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/927>

⁴⁹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/926>

⁴⁹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/925>

⁴⁹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/899>

⁴⁹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/879>

- Issue #865⁴⁹⁸⁰ - future<T> and friends shall work for movable only Ts
- Issue #847⁴⁹⁸¹ - Dynamic libraries are not installed on OS X
- Issue #816⁴⁹⁸² - First Program tutorial pull request
- Issue #799⁴⁹⁸³ - Wrap lexical_cast to avoid exceptions
- Issue #720⁴⁹⁸⁴ - broken configuration when using ccmake on Ubuntu
- Issue #622⁴⁹⁸⁵ - --hpx:hpx and --hpx:debug-hpx-log is nonsensical
- Issue #525⁴⁹⁸⁶ - Extend barrier LCO test to run in distributed
- Issue #515⁴⁹⁸⁷ - Multi-destination version of hpx::apply is broken
- Issue #509⁴⁹⁸⁸ - Push Boost.Atomic changes upstream
- Issue #503⁴⁹⁸⁹ - Running HPX applications on Windows should not require setting %PATH%
- Issue #461⁴⁹⁹⁰ - Add a compilation sanity test
- Issue #456⁴⁹⁹¹ - hpx_run_tests.py should log output from tests that timeout
- Issue #454⁴⁹⁹² - Investigate threadmanager performance
- Issue #345⁴⁹⁹³ - Add more versatile environmental/cmake variable support to hpx_find_* CMake macros
- Issue #209⁴⁹⁹⁴ - Support multiple configurations in generated build files
- Issue #190⁴⁹⁹⁵ - hpx::cout should be a std::ostream
- Issue #189⁴⁹⁹⁶ - iostreams component should use startup/shutdown functions
- Issue #183⁴⁹⁹⁷ - Use Boost.ICL for correctness in AGAS
- Issue #44⁴⁹⁹⁸ - Implement real futures

4980 <https://github.com/STELLAR-GROUP/hpx/issues/865>

4981 <https://github.com/STELLAR-GROUP/hpx/issues/847>

4982 <https://github.com/STELLAR-GROUP/hpx/issues/816>

4983 <https://github.com/STELLAR-GROUP/hpx/issues/799>

4984 <https://github.com/STELLAR-GROUP/hpx/issues/720>

4985 <https://github.com/STELLAR-GROUP/hpx/issues/622>

4986 <https://github.com/STELLAR-GROUP/hpx/issues/525>

4987 <https://github.com/STELLAR-GROUP/hpx/issues/515>

4988 <https://github.com/STELLAR-GROUP/hpx/issues/509>

4989 <https://github.com/STELLAR-GROUP/hpx/issues/503>

4990 <https://github.com/STELLAR-GROUP/hpx/issues/461>

4991 <https://github.com/STELLAR-GROUP/hpx/issues/456>

4992 <https://github.com/STELLAR-GROUP/hpx/issues/454>

4993 <https://github.com/STELLAR-GROUP/hpx/issues/345>

4994 <https://github.com/STELLAR-GROUP/hpx/issues/209>

4995 <https://github.com/STELLAR-GROUP/hpx/issues/190>

4996 <https://github.com/STELLAR-GROUP/hpx/issues/189>

4997 <https://github.com/STELLAR-GROUP/hpx/issues/183>

4998 <https://github.com/STELLAR-GROUP/hpx/issues/44>

2.10.20 HPX V0.9.8 (Mar 24, 2014)

We have had over 800 commits since the last release and we have closed over 65 tickets (bugs, feature requests, etc.).

With the changes below, *HPX* is once again leading the charge of a whole new era of computation. By intrinsically breaking down and synchronizing the work to be done, *HPX* insures that application developers will no longer have to fret about where a segment of code executes. That allows coders to focus their time and energy to understanding the data dependencies of their algorithms and thereby the core obstacles to an efficient code. Here are some of the advantages of using *HPX*:

- *HPX* is solidly rooted in a sophisticated theoretical execution model – ParalleX
- *HPX* exposes an API fully conforming to the C++11 and the draft C++14 standards, extended and applied to distributed computing. Everything programmers know about the concurrency primitives of the standard C++ library is still valid in the context of *HPX*.
- It provides a competitive, high performance implementation of modern, future-proof ideas which gives an smooth migration path from today's mainstream techniques
- There is no need for the programmer to worry about lower level parallelization paradigms like threads or message passing; no need to understand pthreads, MPI, OpenMP, or Windows threads, etc.
- There is no need to think about different types of parallelism such as tasks, pipelines, or fork-join, task or data parallelism.
- The same source of your program compiles and runs on Linux, BlueGene/Q, Mac OS X, Windows, and Android.
- The same code runs on shared memory multi-core systems and supercomputers, on handheld devices and Intel® Xeon Phi™ accelerators, or a heterogeneous mix of those.

General changes

- A major API breaking change for this release was introduced by implementing `hpx::future` and `hpx::shared_future` fully in conformance with the C++11 Standard⁴⁹⁹⁹. 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 archive⁵⁰⁰⁰. 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 Standard⁵⁰⁰¹.
- A large part of the code base of *HPX* has been refactored and partially re-implemented. The main changes were related to
 - The threading subsystem: these changes significantly reduce the amount of overheads caused by the schedulers, improve the modularity of the code base, and extend the variety of available scheduling algorithms.
 - The parcel subsystem: these changes improve the performance of the *HPX* networking layer, modularize the structure of the parcelports, and simplify the creation of new parcelports for other underlying networking libraries.
 - The API subsystem: these changes improved the conformance of the API to C++11 Standard, extend and unify the available API functionality, and decrease the overheads created by various elements of the API.
 - The robustness of the component loading subsystem has been improved significantly, allowing to more portably and more reliably register the components needed by an application as startup. This additionally speeds up general application initialization.

⁴⁹⁹⁹ <http://www.open-std.org/jtc1/sc22/wg21>

⁵⁰⁰⁰ <http://mail.cct.lsu.edu/pipermail/hpx-users/2014-January/000141.html>

⁵⁰⁰¹ <http://www.open-std.org/jtc1/sc22/wg21>

- We added new API functionality like `hpx::migrate` and `hpx::copy_component` which are the basic building blocks necessary for implementing higher level abstractions for system-wide load balancing, runtime-adaptive resource management, and object-oriented checkpointing and state-management.
- We removed the use of C++11 move emulation (using Boost.Move), replacing it with C++11 rvalue references. This is the first step towards using more and more native C++11 facilities which we plan to introduce in the future.
- We improved the reference counting scheme used by *HPX* which helps managing distributed objects and memory. This improves the overall stability of *HPX* and further simplifies writing real world applications.
- The minimal Boost version required to use *HPX* is now V1.49.0.
- This release coincides with the first release of HPXPI (V0.1.0), the first implementation of the XPI specification⁵⁰⁰².

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1086⁵⁰⁰³ - Expose internal boost::shared_array to allow user management of array lifetime
- Issue #1083⁵⁰⁰⁴ - Make shell examples copyable in docs
- Issue #1080⁵⁰⁰⁵ - /threads{locality#*/total}/count/cumulative broken
- Issue #1079⁵⁰⁰⁶ - Build problems on OS X
- Issue #1078⁵⁰⁰⁷ - Improve robustness of component loading
- Issue #1077⁵⁰⁰⁸ - Fix a missing enum definition for ‘take’ mode
- Issue #1076⁵⁰⁰⁹ - Merge Jb master
- Issue #1075⁵⁰¹⁰ - Unknown CMake command “add_hpx_pseudo_target”
- Issue #1074⁵⁰¹¹ - Implement `apply_continue_callback` and `apply_colocated_callback`
- Issue #1073⁵⁰¹² - The new `apply_colocated` and `async_colocated` functions lead to automatic registered functions
- Issue #1071⁵⁰¹³ - Remove deferred_packaged_task
- Issue #1069⁵⁰¹⁴ - `serialize_buffer` with allocator fails at destruction
- Issue #1068⁵⁰¹⁵ - Coroutine include and forward declarations missing
- Issue #1067⁵⁰¹⁶ - Add allocator support to `util::serialize_buffer`

⁵⁰⁰² <https://github.com/STELLAR-GROUP/hpxpi/blob/master/spec.pdf?raw=true>

⁵⁰⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/1086>

⁵⁰⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1083>

⁵⁰⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1080>

⁵⁰⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1079>

⁵⁰⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1078>

⁵⁰⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1077>

⁵⁰⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1076>

⁵⁰¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1075>

⁵⁰¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/1074>

⁵⁰¹² <https://github.com/STELLAR-GROUP/hpx/issues/1073>

⁵⁰¹³ <https://github.com/STELLAR-GROUP/hpx/issues/1071>

⁵⁰¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1069>

⁵⁰¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1068>

⁵⁰¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1067>

- Issue #1066⁵⁰¹⁷ - Allow for MPI_Init being called before HPX launches
- Issue #1065⁵⁰¹⁸ - AGAS cache isn't used/populated on worker localities
- Issue #1064⁵⁰¹⁹ - Reorder includes to ensure ws2 includes early
- Issue #1063⁵⁰²⁰ - Add hpx::runtime::suspend and hpx::runtime::resume
- Issue #1062⁵⁰²¹ - Fix async_continue to properly handle return types
- Issue #1061⁵⁰²² - Implement async_colocated and apply_colocated
- Issue #1060⁵⁰²³ - Implement minimal component migration
- Issue #1058⁵⁰²⁴ - Remove HPX_UTIL_TUPLE from code base
- Issue #1057⁵⁰²⁵ - Add performance counters for threading subsystem
- Issue #1055⁵⁰²⁶ - Thread allocation uses two memory pools
- Issue #1053⁵⁰²⁷ - Work stealing flawed
- Issue #1052⁵⁰²⁸ - Fix a number of warnings
- Issue #1049⁵⁰²⁹ - Fixes for TLS on OSX and more reliable test running
- Issue #1048⁵⁰³⁰ - Fixing after 588 hang
- Issue #1047⁵⁰³¹ - Use port '0' for networking when using one locality
- Issue #1046⁵⁰³² - composable_guard test is broken when having more than one thread
- Issue #1045⁵⁰³³ - Security missing headers
- Issue #1044⁵⁰³⁴ - Native TLS on FreeBSD via __thread
- Issue #1043⁵⁰³⁵ - async et.al. compute the wrong result type
- Issue #1042⁵⁰³⁶ - async et.al. implicitly unwrap reference_wrappers
- Issue #1041⁵⁰³⁷ - Remove redundant costly Kleene stars from regex searches
- Issue #1040⁵⁰³⁸ - CMake script regex match patterns has unnecessary kleenes
- Issue #1039⁵⁰³⁹ - Remove use of Boost.Move and replace with std::move and real rvalue refs

⁵⁰¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1066>

⁵⁰¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1065>

⁵⁰¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1064>

⁵⁰²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1063>

⁵⁰²¹ <https://github.com/STELLAR-GROUP/hpx/issues/1062>

⁵⁰²² <https://github.com/STELLAR-GROUP/hpx/issues/1061>

⁵⁰²³ <https://github.com/STELLAR-GROUP/hpx/issues/1060>

⁵⁰²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1058>

⁵⁰²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1057>

⁵⁰²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1055>

⁵⁰²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1053>

⁵⁰²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1052>

⁵⁰²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1049>

⁵⁰³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1048>

⁵⁰³¹ <https://github.com/STELLAR-GROUP/hpx/issues/1047>

⁵⁰³² <https://github.com/STELLAR-GROUP/hpx/issues/1046>

⁵⁰³³ <https://github.com/STELLAR-GROUP/hpx/issues/1045>

⁵⁰³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1044>

⁵⁰³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1043>

⁵⁰³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1042>

⁵⁰³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1041>

⁵⁰³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1040>

⁵⁰³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1039>

- Issue #1038⁵⁰⁴⁰ - Bump minimal required Boost to 1.49.0
- Issue #1037⁵⁰⁴¹ - Implicit unwrapping of futures in async broken
- Issue #1036⁵⁰⁴² - Scheduler hangs when user code attempts to “block” OS-threads
- Issue #1035⁵⁰⁴³ - Idle-rate counter always reports 100% idle rate
- Issue #1034⁵⁰⁴⁴ - Symbolic name registration causes application hangs
- Issue #1033⁵⁰⁴⁵ - Application options read in from an options file generate an error message
- Issue #1032⁵⁰⁴⁶ - `hpx::id_type` local reference counting is wrong
- Issue #1031⁵⁰⁴⁷ - Negative entry in reference count table
- Issue #1030⁵⁰⁴⁸ - Implement condition_variable
- Issue #1029⁵⁰⁴⁹ - Deadlock in thread scheduling subsystem
- Issue #1028⁵⁰⁵⁰ - HPX-thread cumulative count performance counters report incorrect value
- Issue #1027⁵⁰⁵¹ - Expose `hpx::thread_interrupted` error code as a separate exception type
- Issue #1026⁵⁰⁵² - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
- Issue #1025⁵⁰⁵³ - `future::wait_for/wait_until` do not remove callback
- Issue #1024⁵⁰⁵⁴ - Remove dependence to boost assert and create hpx assert
- Issue #1023⁵⁰⁵⁵ - Segfaults with tcmalloc
- Issue #1022⁵⁰⁵⁶ - prerequisites link in readme is broken
- Issue #1020⁵⁰⁵⁷ - HPX Deadlock on external synchronization
- Issue #1019⁵⁰⁵⁸ - Convert using BOOST_ASSERT to HPX_ASSERT
- Issue #1018⁵⁰⁵⁹ - compiling bug with gcc 4.8.1
- Issue #1017⁵⁰⁶⁰ - Possible crash in io_pool executor
- Issue #1016⁵⁰⁶¹ - Crash at startup
- Issue #1014⁵⁰⁶² - Implement Increment/Decrement Merging

⁵⁰⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1038>

⁵⁰⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/1037>

⁵⁰⁴² <https://github.com/STELLAR-GROUP/hpx/issues/1036>

⁵⁰⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/1035>

⁵⁰⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1034>

⁵⁰⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1033>

⁵⁰⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1032>

⁵⁰⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1031>

⁵⁰⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1030>

⁵⁰⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1029>

⁵⁰⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1028>

⁵⁰⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/1027>

⁵⁰⁵² <https://github.com/STELLAR-GROUP/hpx/issues/1026>

⁵⁰⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/1025>

⁵⁰⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1024>

⁵⁰⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1023>

⁵⁰⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1022>

⁵⁰⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1020>

⁵⁰⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1019>

⁵⁰⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1018>

⁵⁰⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1017>

⁵⁰⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/1016>

⁵⁰⁶² <https://github.com/STELLAR-GROUP/hpx/issues/1014>

- Issue #1013⁵⁰⁶³ - Add more logging channels to enable greater control over logging granularity
- Issue #1012⁵⁰⁶⁴ - --hpx:debug-hpx-log and --hpx:debug-agas-log lead to non-thread safe writes
- Issue #1011⁵⁰⁶⁵ - After installation, running applications from the build/staging directory no longer works
- Issue #1010⁵⁰⁶⁶ - Mergeable decrement requests are not being merged
- Issue #1009⁵⁰⁶⁷ - --hpx:list-symbolic-names crashes
- Issue #1007⁵⁰⁶⁸ - Components are not properly destroyed
- Issue #1006⁵⁰⁶⁹ - Segfault/hang in set_data
- Issue #1003⁵⁰⁷⁰ - Performance counter naming issue
- Issue #982⁵⁰⁷¹ - Race condition during startup
- Issue #912⁵⁰⁷² - OS X: component type not found in map
- Issue #663⁵⁰⁷³ - Create a buildbot slave based on Clang 3.2/OSX
- Issue #636⁵⁰⁷⁴ - Expose `this_locality::apply<act>(p1, p2);` for local execution
- Issue #197⁵⁰⁷⁵ - Add --console=address option for PBS runs
- Issue #175⁵⁰⁷⁶ - Asynchronous AGAS API

2.10.21 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

⁵⁰⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/1013>

⁵⁰⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/1012>

⁵⁰⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/1011>

⁵⁰⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/1010>

⁵⁰⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1009>

⁵⁰⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1007>

⁵⁰⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1006>

⁵⁰⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1003>

⁵⁰⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/982>

⁵⁰⁷² <https://github.com/STELLAR-GROUP/hpx/issues/912>

⁵⁰⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/663>

⁵⁰⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/636>

⁵⁰⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/197>

⁵⁰⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/175>

- 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 #1005⁵⁰⁷⁷ - Allow one to disable array optimizations and zero copy optimizations for each parcelport
- Issue #1004⁵⁰⁷⁸ - Generate new HPX logo image for the docs
- Issue #1002⁵⁰⁷⁹ - If MPI parcelport is not available, running HPX under mpirun should fail
- Issue #1001⁵⁰⁸⁰ - Zero copy serialization raises assert
- Issue #1000⁵⁰⁸¹ - Can't connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
- Issue #999⁵⁰⁸² - Optimize `hpx::when_n`
- Issue #998⁵⁰⁸³ - Fixed const-correctness
- Issue #997⁵⁰⁸⁴ - Making `serialize_buffer::data()` type save
- Issue #996⁵⁰⁸⁵ - Memory leak in `hpx::lcos::promise`
- Issue #995⁵⁰⁸⁶ - Race while registering pre-shutdown functions
- Issue #994⁵⁰⁸⁷ - `thread_rescheduling` regression test does not compile
- Issue #992⁵⁰⁸⁸ - Correct comments and messages
- Issue #991⁵⁰⁸⁹ - `setcap cap_sys_rawio=ep` for power profiling causes an HPX application to abort
- Issue #989⁵⁰⁹⁰ - Jacobi hangs during execution
- Issue #988⁵⁰⁹¹ - `multiple_init` test is failing
- Issue #986⁵⁰⁹² - Can't call a function called "init" from "main" when using `<hpx/hpx_main.hpp>`

⁵⁰⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/1005>

⁵⁰⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/1004>

⁵⁰⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/1002>

⁵⁰⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/1001>

⁵⁰⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/1000>

⁵⁰⁸² <https://github.com/STELLAR-GROUP/hpx/issues/999>

⁵⁰⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/998>

⁵⁰⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/997>

⁵⁰⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/996>

⁵⁰⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/995>

⁵⁰⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/994>

⁵⁰⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/992>

⁵⁰⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/991>

⁵⁰⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/989>

⁵⁰⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/988>

⁵⁰⁹² <https://github.com/STELLAR-GROUP/hpx/issues/986>

- Issue #984⁵⁰⁹³ - Reference counting tests are failing
- Issue #983⁵⁰⁹⁴ - thread_suspension_executor test fails
- Issue #980⁵⁰⁹⁵ - Terminating HPX threads don't leave stack in virgin state
- Issue #979⁵⁰⁹⁶ - Static scheduler not in documents
- Issue #978⁵⁰⁹⁷ - Preprocessing limits are broken
- Issue #977⁵⁰⁹⁸ - Make tests.regressions.lcos.future_hang_on_get shorter
- Issue #976⁵⁰⁹⁹ - Wrong library order in pkgconfig
- Issue #975⁵¹⁰⁰ - Please reopen #963
- Issue #974⁵¹⁰¹ - Option pu-offset ignored in fixing_588 branch
- Issue #972⁵¹⁰² - Cannot use MKL with HPX
- Issue #969⁵¹⁰³ - Non-existent INI files requested on the command line via --hpx:config do not cause warnings or errors.
- Issue #968⁵¹⁰⁴ - Cannot build examples in fixing_588 branch
- Issue #967⁵¹⁰⁵ - Command line description of --hpx:queuing seems wrong
- Issue #966⁵¹⁰⁶ - --hpx:print-bind physical core numbers are wrong
- Issue #965⁵¹⁰⁷ - Deadlock when building in Release mode
- Issue #963⁵¹⁰⁸ - Not all worker threads are working
- Issue #962⁵¹⁰⁹ - Problem with SLURM integration
- Issue #961⁵¹¹⁰ - --hpx:print-bind outputs incorrect information
- Issue #960⁵¹¹¹ - Fix cut and paste error in documentation of get_thread_priority
- Issue #959⁵¹¹² - Change link to boost.atomic in documentation to point to boost.org
- Issue #958⁵¹¹³ - Undefined reference to intrusive_ptr_release
- Issue #957⁵¹¹⁴ - Make tuple standard compliant
- Issue #956⁵¹¹⁵ - Segfault with a3382fb

⁵⁰⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/984>

⁵⁰⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/983>

⁵⁰⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/980>

⁵⁰⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/979>

⁵⁰⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/978>

⁵⁰⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/977>

⁵⁰⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/976>

⁵¹⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/975>

⁵¹⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/974>

⁵¹⁰² <https://github.com/STELLAR-GROUP/hpx/issues/972>

⁵¹⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/969>

⁵¹⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/968>

⁵¹⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/967>

⁵¹⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/966>

⁵¹⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/965>

⁵¹⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/963>

⁵¹⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/962>

⁵¹¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/961>

⁵¹¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/960>

⁵¹¹² <https://github.com/STELLAR-GROUP/hpx/issues/959>

⁵¹¹³ <https://github.com/STELLAR-GROUP/hpx/issues/958>

⁵¹¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/957>

⁵¹¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/956>

- Issue #955⁵¹¹⁶ --hpx:nodes and --hpx:nodenames do not work with foreign nodes
- Issue #954⁵¹¹⁷ - Make order of arguments for hpx::async and hpx::broadcast consistent
- Issue #953⁵¹¹⁸ - Cannot use MKL with HPX
- Issue #952⁵¹¹⁹ - register_[pre_]shutdown_function never throw
- Issue #951⁵¹²⁰ - Assert when number of threads is greater than hardware concurrency
- Issue #948⁵¹²¹ - HPX_HAVE_GENERIC_CONTEXT_COROUTINES conflicts with HPX_HAVE_FIBER_BASED_COROUTINES
- Issue #947⁵¹²² - Need MPI_THREAD_MULTIPLE for backward compatibility
- Issue #946⁵¹²³ - HPX does not call MPI_Finalize
- Issue #945⁵¹²⁴ - Segfault with hpx::lcos::broadcast
- Issue #944⁵¹²⁵ - OS X: assertion pu_offset_ < hardware_concurrency failed
- Issue #943⁵¹²⁶ - #include <hpx/hpx_main.hpp> does not work
- Issue #942⁵¹²⁷ - Make the BG/Q work with -O3
- Issue #940⁵¹²⁸ - Use separator when concatenating locality name
- Issue #939⁵¹²⁹ - Refactor MPI parcelport to use MPI_Wait instead of multiple MPI_Test calls
- Issue #938⁵¹³⁰ - Want to officially access client_base::gid_
- Issue #937⁵¹³¹ - client_base::gid_ should be private``
- Issue #936⁵¹³² - Want doxygen-like source code index
- Issue #935⁵¹³³ - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #933⁵¹³⁴ - Cannot build HPX with Boost 1.54.0
- Issue #932⁵¹³⁵ - Components are destructed too early
- Issue #931⁵¹³⁶ - Make HPX work on BG/Q
- Issue #930⁵¹³⁷ - make git-docs is broken
- Issue #929⁵¹³⁸ - Generating index in docs broken

⁵¹¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/955>

⁵¹¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/954>

⁵¹¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/953>

⁵¹¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/952>

⁵¹²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/951>

⁵¹²¹ <https://github.com/STELLAR-GROUP/hpx/issues/948>

⁵¹²² <https://github.com/STELLAR-GROUP/hpx/issues/947>

⁵¹²³ <https://github.com/STELLAR-GROUP/hpx/issues/946>

⁵¹²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/945>

⁵¹²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/944>

⁵¹²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/943>

⁵¹²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/942>

⁵¹²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/940>

⁵¹²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/939>

⁵¹³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/938>

⁵¹³¹ <https://github.com/STELLAR-GROUP/hpx/issues/937>

⁵¹³² <https://github.com/STELLAR-GROUP/hpx/issues/936>

⁵¹³³ <https://github.com/STELLAR-GROUP/hpx/issues/935>

⁵¹³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/933>

⁵¹³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/932>

⁵¹³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/931>

⁵¹³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/930>

⁵¹³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/929>

- Issue #928⁵¹³⁹ - Optimize `hpx::util::static_` for C++11 compilers supporting magic statics
- Issue #924⁵¹⁴⁰ - Make `kill_process_tree` (in `process.py`) more robust on Mac OSX
- Issue #923⁵¹⁴¹ - Correct BLAS and RNPL cmake tests
- Issue #922⁵¹⁴² - Cannot link against BLAS
- Issue #921⁵¹⁴³ - Implement `hpx::mem_fn`
- Issue #920⁵¹⁴⁴ - Output locality with `--hpx:print-bind`
- Issue #919⁵¹⁴⁵ - Correct grammar; simplify boolean expressions
- Issue #918⁵¹⁴⁶ - Link to `hello_world.cpp` is broken
- Issue #917⁵¹⁴⁷ - adapt cmake file to new boostbook version
- Issue #916⁵¹⁴⁸ - fix problem building documentation with `xsltproc >= 1.1.27`
- Issue #915⁵¹⁴⁹ - Add another TBBMalloc library search path
- Issue #914⁵¹⁵⁰ - Build problem with Intel compiler on Stampede (TACC)
- Issue #913⁵¹⁵¹ - fix error messages in fibonacci examples
- Issue #911⁵¹⁵² - Update OS X build instructions
- Issue #910⁵¹⁵³ - Want like to specify `MPI_ROOT` instead of compiler wrapper script
- Issue #909⁵¹⁵⁴ - Warning about `void*` arithmetic
- Issue #908⁵¹⁵⁵ - Buildbot for MIC is broken
- Issue #906⁵¹⁵⁶ - Can't use `--hpx:bind=balanced` with multiple MPI processes
- Issue #905⁵¹⁵⁷ - `--hpx:bind` documentation should describe full grammar
- Issue #904⁵¹⁵⁸ - Add `hpx::lcos::fold` and `hpx::lcos::inverse_fold` collective operation
- Issue #903⁵¹⁵⁹ - Add `hpx::when_any_swapped()`
- Issue #902⁵¹⁶⁰ - Add `hpx::lcos::reduce` collective operation
- Issue #901⁵¹⁶¹ - Web documentation is not searchable

⁵¹³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/928>

⁵¹⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/924>

⁵¹⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/923>

⁵¹⁴² <https://github.com/STELLAR-GROUP/hpx/issues/922>

⁵¹⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/921>

⁵¹⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/920>

⁵¹⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/919>

⁵¹⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/918>

⁵¹⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/917>

⁵¹⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/916>

⁵¹⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/915>

⁵¹⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/914>

⁵¹⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/913>

⁵¹⁵² <https://github.com/STELLAR-GROUP/hpx/issues/911>

⁵¹⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/910>

⁵¹⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/909>

⁵¹⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/908>

⁵¹⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/906>

⁵¹⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/905>

⁵¹⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/904>

⁵¹⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/903>

⁵¹⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/902>

⁵¹⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/901>

- Issue #900⁵¹⁶² - Web documentation for trunk has no index
- Issue #898⁵¹⁶³ - Some tests fail with GCC 4.8.1 and MPI parcel port
- Issue #897⁵¹⁶⁴ - HWLOC causes failures on Mac
- Issue #896⁵¹⁶⁵ - pu-offset leads to startup error
- Issue #895⁵¹⁶⁶ - `hpx::get_locality_name` not defined
- Issue #894⁵¹⁶⁷ - Race condition at shutdown
- Issue #893⁵¹⁶⁸ - --`hpx:print-bind` switches std::cout to hexadecimal mode
- Issue #892⁵¹⁶⁹ - `hwloc_topology_load` can be expensive – don't call multiple times
- Issue #891⁵¹⁷⁰ - The documentation for `get_locality_name` is wrong
- Issue #890⁵¹⁷¹ - --`hpx:print-bind` should not exit
- Issue #889⁵¹⁷² - --`hpx:debug-hpx-log=FILE` does not work
- Issue #888⁵¹⁷³ - MPI parcelport does not exit cleanly for --`hpx:print-bind`
- Issue #887⁵¹⁷⁴ - Choose thread affinities more cleverly
- Issue #886⁵¹⁷⁵ - Logging documentation is confusing
- Issue #885⁵¹⁷⁶ - Two threads are slower than one
- Issue #884⁵¹⁷⁷ - `is_callable` failing with member pointers in C++11
- Issue #883⁵¹⁷⁸ - Need help with `is_callable_test`
- Issue #882⁵¹⁷⁹ - tests/regressions.lcos.future_hang_on_get does not terminate
- Issue #881⁵¹⁸⁰ - tests/regressions/block_matrix/matrix.hh won't compile with GCC 4.8.1
- Issue #880⁵¹⁸¹ - HPX does not work on OS X
- Issue #878⁵¹⁸² - `future::unwrap` triggers assertion
- Issue #877⁵¹⁸³ - “make tests” has build errors on Ubuntu 12.10
- Issue #876⁵¹⁸⁴ - `tcmalloc` is used by default, even if it is not present

⁵¹⁶² <https://github.com/STELLAR-GROUP/hpx/issues/900>

⁵¹⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/898>

⁵¹⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/897>

⁵¹⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/896>

⁵¹⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/895>

⁵¹⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/894>

⁵¹⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/893>

⁵¹⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/892>

⁵¹⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/891>

⁵¹⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/890>

⁵¹⁷² <https://github.com/STELLAR-GROUP/hpx/issues/889>

⁵¹⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/888>

⁵¹⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/887>

⁵¹⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/886>

⁵¹⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/885>

⁵¹⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/884>

⁵¹⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/883>

⁵¹⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/882>

⁵¹⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/881>

⁵¹⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/880>

⁵¹⁸² <https://github.com/STELLAR-GROUP/hpx/issues/878>

⁵¹⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/877>

⁵¹⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/876>

- Issue #875⁵¹⁸⁵ - global_fixture is defined in a header file
- Issue #874⁵¹⁸⁶ - Some tests take very long
- Issue #873⁵¹⁸⁷ - Add block-matrix code as regression test
- Issue #872⁵¹⁸⁸ - HPX documentation does not say how to run tests with detailed output
- Issue #871⁵¹⁸⁹ - All tests fail with “make test”
- Issue #870⁵¹⁹⁰ - Please explicitly disable serialization in classes that don’t support it
- Issue #868⁵¹⁹¹ - boost_any test failing
- Issue #867⁵¹⁹² - Reduce the number of copies of `hpx::function` arguments
- Issue #863⁵¹⁹³ - Futures should not require a default constructor
- Issue #862⁵¹⁹⁴ - `value_or_error` shall not default construct its result
- Issue #861⁵¹⁹⁵ - `HPX_UNUSED` macro
- Issue #860⁵¹⁹⁶ - Add functionality to copy construct a component
- Issue #859⁵¹⁹⁷ - `hpx::endl` should flush
- Issue #858⁵¹⁹⁸ - Create `hpx::get_ptr<>` allowing to access component implementation
- Issue #855⁵¹⁹⁹ - Implement `hpx::(INVOKE`
- Issue #854⁵²⁰⁰ - `hpx/hpx.hpp` does not include `hpx/include/iostreams.hpp`
- Issue #853⁵²⁰¹ - Feature request: null future
- Issue #852⁵²⁰² - Feature request: Locality names
- Issue #851⁵²⁰³ - `hpx::cout` output does not appear on screen
- Issue #849⁵²⁰⁴ - All tests fail on OS X after installing
- Issue #848⁵²⁰⁵ - Update OS X build instructions
- Issue #846⁵²⁰⁶ - Update `hpx_external_example`
- Issue #845⁵²⁰⁷ - Issues with having both debug and release modules in the same directory

⁵¹⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/875>

⁵¹⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/874>

⁵¹⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/873>

⁵¹⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/872>

⁵¹⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/871>

⁵¹⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/870>

⁵¹⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/868>

⁵¹⁹² <https://github.com/STELLAR-GROUP/hpx/issues/867>

⁵¹⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/863>

⁵¹⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/862>

⁵¹⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/861>

⁵¹⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/860>

⁵¹⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/859>

⁵¹⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/858>

⁵¹⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/855>

⁵²⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/854>

⁵²⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/853>

⁵²⁰² <https://github.com/STELLAR-GROUP/hpx/issues/852>

⁵²⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/851>

⁵²⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/849>

⁵²⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/848>

⁵²⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/846>

⁵²⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/845>

- Issue #844⁵²⁰⁸ - Create configuration header
- Issue #843⁵²⁰⁹ - Tests should use CTest
- Issue #842⁵²¹⁰ - Remove buffer_pool from MPI parcelport
- Issue #841⁵²¹¹ - Add possibility to broadcast an index with hpx::lcos::broadcast
- Issue #838⁵²¹² - Simplify util::tuple
- Issue #837⁵²¹³ - Adopt boost::tuple tests for util::tuple
- Issue #836⁵²¹⁴ - Adopt boost::function tests for util::function
- Issue #835⁵²¹⁵ - Tuple interface missing pieces
- Issue #833⁵²¹⁶ - Partially preprocessing files not working
- Issue #832⁵²¹⁷ - Native papi counters do not work with wild cards
- Issue #831⁵²¹⁸ - Arithmetics counter fails if only one parameter is given
- Issue #830⁵²¹⁹ - Convert hpx::util::function to use new scheme for serializing its base pointer
- Issue #829⁵²²⁰ - Consistently use decay<T> instead of remove_const< remove_reference<T>>
- Issue #828⁵²²¹ - Update future implementation to N3721 and N3722
- Issue #827⁵²²² - Enable MPI parcelport for bootstrapping whenever application was started using mpirun
- Issue #826⁵²²³ - Support command line option --hpx:print-bind even if --hpx::bind was not used
- Issue #825⁵²²⁴ - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- Issue #824⁵²²⁵ - Enable lambda functions to be used with hpx::async/hpx::apply
- Issue #823⁵²²⁶ - Using a hashing filter
- Issue #822⁵²²⁷ - Silence unused variable warning
- Issue #821⁵²²⁸ - Detect if a function object is callable with given arguments
- Issue #820⁵²²⁹ - Allow wildcards to be used for performance counter names
- Issue #819⁵²³⁰ - Make the AGAS symbolic name registry distributed

⁵²⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/844>

⁵²⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/843>

⁵²¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/842>

⁵²¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/841>

⁵²¹² <https://github.com/STELLAR-GROUP/hpx/issues/838>

⁵²¹³ <https://github.com/STELLAR-GROUP/hpx/issues/837>

⁵²¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/836>

⁵²¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/835>

⁵²¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/833>

⁵²¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/832>

⁵²¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/831>

⁵²¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/830>

⁵²²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/829>

⁵²²¹ <https://github.com/STELLAR-GROUP/hpx/issues/828>

⁵²²² <https://github.com/STELLAR-GROUP/hpx/issues/827>

⁵²²³ <https://github.com/STELLAR-GROUP/hpx/issues/826>

⁵²²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/825>

⁵²²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/824>

⁵²²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/823>

⁵²²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/822>

⁵²²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/821>

⁵²²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/820>

⁵²³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/819>

- Issue #818⁵²³¹ - Add future::then() overload taking an executor
- Issue #817⁵²³² - Fixed typo
- Issue #815⁵²³³ - Create an lco that is performing an efficient broadcast of actions
- Issue #814⁵²³⁴ - Papi counters cannot specify thread#* to get the counts for all threads
- Issue #813⁵²³⁵ - Scoped unlock
- Issue #811⁵²³⁶ - simple_central_tuplespace_client run error
- Issue #810⁵²³⁷ - ostream error when << any objects
- Issue #809⁵²³⁸ - Optimize parcel serialization
- Issue #808⁵²³⁹ - HPX applications throw exception when executed from the build directory
- Issue #807⁵²⁴⁰ - Create performance counters exposing overall AGAS statistics
- Issue #795⁵²⁴¹ - Create timed make_ready_future
- Issue #794⁵²⁴² - Create heterogeneous when_all/when_any/etc.
- Issue #721⁵²⁴³ - Make HPX usable for Xeon Phi
- Issue #694⁵²⁴⁴ - CMake should complain if you attempt to build an example without its dependencies
- Issue #692⁵²⁴⁵ - SLURM support broken
- Issue #683⁵²⁴⁶ - python/hpx/process.py imports epoll on all platforms
- Issue #619⁵²⁴⁷ - Automate the doc building process
- Issue #600⁵²⁴⁸ - GTC performance broken
- Issue #577⁵²⁴⁹ - Allow for zero copy serialization/networking
- Issue #551⁵²⁵⁰ - Change executable names to have debug postfix in Debug builds
- Issue #544⁵²⁵¹ - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init
- Issue #534⁵²⁵² - hpx::init should take functions by std::function and should accept all forms of hpx_main
- Issue #508⁵²⁵³ - FindPackage fails to set FOO_LIBRARY_DIR

⁵²³¹ <https://github.com/STELLAR-GROUP/hpx/issues/818>

⁵²³² <https://github.com/STELLAR-GROUP/hpx/issues/817>

⁵²³³ <https://github.com/STELLAR-GROUP/hpx/issues/815>

⁵²³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/814>

⁵²³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/813>

⁵²³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/811>

⁵²³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/810>

⁵²³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/809>

⁵²³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/808>

⁵²⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/807>

⁵²⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/795>

⁵²⁴² <https://github.com/STELLAR-GROUP/hpx/issues/794>

⁵²⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/721>

⁵²⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/694>

⁵²⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/692>

⁵²⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/683>

⁵²⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/619>

⁵²⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/600>

⁵²⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/577>

⁵²⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/551>

⁵²⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/544>

⁵²⁵² <https://github.com/STELLAR-GROUP/hpx/issues/534>

⁵²⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/508>

- Issue #506⁵²⁵⁴ - Add cmake support to generate ini files for external applications
- Issue #470⁵²⁵⁵ - Changing build-type after configure does not update boost library names
- Issue #453⁵²⁵⁶ - Document hpx_run_tests.py
- Issue #445⁵²⁵⁷ - Significant performance mismatch between MPI and HPX in SMP for allgather example
- Issue #443⁵²⁵⁸ - Make docs viewable from build directory
- Issue #421⁵²⁵⁹ - Support multiple HPX instances per node in a batch environment like PBS or SLURM
- Issue #316⁵²⁶⁰ - Add message size limitation
- Issue #249⁵²⁶¹ - Clean up locking code in big boot barrier
- Issue #136⁵²⁶² - Persistent CMake variables need to be marked as cache variables

2.10.22 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 Standard⁵²⁶³ and related proposals to the C++ standardization committee (such as N3632⁵²⁶⁴ and N3857⁵²⁶⁵).
- 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.

⁵²⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/506>

⁵²⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/470>

⁵²⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/453>

⁵²⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/445>

⁵²⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/443>

⁵²⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/421>

⁵²⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/316>

⁵²⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/249>

⁵²⁶² <https://github.com/STELLAR-GROUP/hpx/issues/136>

⁵²⁶³ <http://www.open-std.org/jtc1/sc22/wg21>

⁵²⁶⁴ <http://wg21.link/n3632>

⁵²⁶⁵ <http://wg21.link/n3857>

- Added an experimental executors API allowing to use different scheduling policies for different parts of the code. This API has been modelled after the Standards proposal [N3562](#)⁵²⁶⁶. This API is bound to change in the future, though.
- Added minimal security support for localities which is enforced on the parcelport level. This support is preliminary and experimental and might change in the future.
- We created a parcelport using low level MPI functions. This is in support of legacy applications which are to be gradually ported and to support platforms where MPI is the only available portable networking layer.
- We added a preliminary and experimental implementation of a tuple-space object which exposes an interface similar to such systems described in the literature (see for instance [The Linda Coordination Language](#)⁵²⁶⁷).

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 #806⁵²⁶⁸ - make (all) in examples folder does nothing
- Issue #805⁵²⁶⁹ - Adding the introduction and fixing DOCBOOK dependencies for Windows use
- Issue #804⁵²⁷⁰ - Add stackless (non-suspendable) thread type
- Issue #803⁵²⁷¹ - Create proper serialization support functions for util::tuple
- Issue #800⁵²⁷² - Add possibility to disable array optimizations during serialization
- Issue #798⁵²⁷³ - HPX_LIMIT does not work for local dataflow
- Issue #797⁵²⁷⁴ - Create a parcelport which uses MPI
- Issue #796⁵²⁷⁵ - Problem with Large Numbers of Threads
- Issue #793⁵²⁷⁶ - Changing dataflow test case to hang consistently
- Issue #792⁵²⁷⁷ - CMake Error
- Issue #791⁵²⁷⁸ - Problems with local::dataflow
- Issue #790⁵²⁷⁹ - wait_for() doesn't compile
- Issue #789⁵²⁸⁰ - HPX with Intel compiler segfaults
- Issue #788⁵²⁸¹ - Intel compiler support
- Issue #787⁵²⁸² - Fixed SFINAEd specializations

⁵²⁶⁶ <http://wg21.link/n3562>

⁵²⁶⁷ [https://en.wikipedia.org/wiki/Linda_\(coordination_language\)](https://en.wikipedia.org/wiki/Linda_(coordination_language))

⁵²⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/806>

5269 <https://github.com/STELLAR-GROUP/hpx/issues/805>5270 <https://github.com/STELLAR-GROUP/hpx/issues/804>5271 <https://github.com/STELLAR-GROUP/hpx/issues/803>5272 <https://github.com/STELLAR-GROUP/hpx/issues/800>5273 <https://github.com/STELLAR-GROUP/hpx/issues/798>5274 <https://github.com/STELLAR-GROUP/hpx/issues/797>5275 <https://github.com/STELLAR-GROUP/hpx/issues/796>5276 <https://github.com/STELLAR-GROUP/hpx/issues/793>5277 <https://github.com/STELLAR-GROUP/hpx/issues/792>5278 <https://github.com/STELLAR-GROUP/hpx/issues/791>5279 <https://github.com/STELLAR-GROUP/hpx/issues/790>5280 <https://github.com/STELLAR-GROUP/hpx/issues/789>5281 <https://github.com/STELLAR-GROUP/hpx/issues/788>5282 <https://github.com/STELLAR-GROUP/hpx/issues/787>

- Issue #786⁵²⁸³ - Memory issues during benchmarking.
- Issue #785⁵²⁸⁴ - Create an API allowing to register external threads with HPX
- Issue #784⁵²⁸⁵ - util::plugin is throwing an error when a symbol is not found
- Issue #783⁵²⁸⁶ - How does hpx:bind work?
- Issue #782⁵²⁸⁷ - Added quotes around STRING REPLACE potentially empty arguments
- Issue #781⁵²⁸⁸ - Make sure no exceptions propagate into the thread manager
- Issue #780⁵²⁸⁹ - Allow arithmetics performance counters to expand its parameters
- Issue #779⁵²⁹⁰ - Test case for 778
- Issue #778⁵²⁹¹ - Swapping futures segfaults
- Issue #777⁵²⁹² - hpx::lcos::details::when_xxx don't restore completion handlers
- Issue #776⁵²⁹³ - Compiler chokes on dataflow overload with launch policy
- Issue #775⁵²⁹⁴ - Runtime error with local dataflow (copying futures?)
- Issue #774⁵²⁹⁵ - Using local dataflow without explicit namespace
- Issue #773⁵²⁹⁶ - Local dataflow with unwrap: functor operators need to be const
- Issue #772⁵²⁹⁷ - Allow (remote) actions to return a future
- Issue #771⁵²⁹⁸ - Setting HPX_LIMIT gives huge boost MPL errors
- Issue #770⁵²⁹⁹ - Add launch policy to (local) dataflow
- Issue #769⁵³⁰⁰ - Make compile time configuration information available
- Issue #768⁵³⁰¹ - Const correctness problem in local dataflow
- Issue #767⁵³⁰² - Add launch policies to async
- Issue #766⁵³⁰³ - Mark data structures for optimized (array based) serialization
- Issue #765⁵³⁰⁴ - Align hpx::any with N3508: Any Library Proposal (Revision 2)
- Issue #764⁵³⁰⁵ - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations

⁵²⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/786>

⁵²⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/785>

⁵²⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/784>

⁵²⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/783>

⁵²⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/782>

⁵²⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/781>

⁵²⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/780>

⁵²⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/779>

⁵²⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/778>

⁵²⁹² <https://github.com/STELLAR-GROUP/hpx/issues/777>

⁵²⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/776>

⁵²⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/775>

⁵²⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/774>

⁵²⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/773>

⁵²⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/772>

⁵²⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/771>

⁵²⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/770>

⁵³⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/769>

⁵³⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/768>

⁵³⁰² <https://github.com/STELLAR-GROUP/hpx/issues/767>

⁵³⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/766>

⁵³⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/765>

⁵³⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/764>

- Issue #762⁵³⁰⁶ - added a human readable output for the ping pong example
- Issue #761⁵³⁰⁷ - Ambiguous typename when constructing derived component
- Issue #760⁵³⁰⁸ - Simple components can not be derived
- Issue #759⁵³⁰⁹ - make install doesn't give a complete install
- Issue #758⁵³¹⁰ - Stack overflow when using locking_hook<>
- Issue #757⁵³¹¹ - copy paste error; unsupported function overloading
- Issue #756⁵³¹² - GTCX runtime issue in Gordon
- Issue #755⁵³¹³ - Papi counters don't work with reset and evaluate API's
- Issue #753⁵³¹⁴ - cmake bugfix and improved component action docs
- Issue #752⁵³¹⁵ - hpx simple component docs
- Issue #750⁵³¹⁶ - Add hpx::util::any
- Issue #749⁵³¹⁷ - Thread phase counter is not reset
- Issue #748⁵³¹⁸ - Memory performance counter are not registered
- Issue #747⁵³¹⁹ - Create performance counters exposing arithmetic operations
- Issue #745⁵³²⁰ - apply_callback needs to invoke callback when applied locally
- Issue #744⁵³²¹ - CMake fixes
- Issue #743⁵³²² - Problem Building github version of HPX
- Issue #742⁵³²³ - Remove HPX_STD_BIND
- Issue #741⁵³²⁴ - assertion 'px != 0' failed: HPX(assertion_failure) for low numbers of OS threads
- Issue #739⁵³²⁵ - Performance counters do not count to the end of the program or evaluation
- Issue #738⁵³²⁶ - Dedicated AGAS server runs don't work; console ignores -a option.
- Issue #737⁵³²⁷ - Missing bind overloads
- Issue #736⁵³²⁸ - Performance counter wildcards do not always work

⁵³⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/762>

⁵³⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/761>

⁵³⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/760>

⁵³⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/759>

⁵³¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/758>

⁵³¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/757>

⁵³¹² <https://github.com/STELLAR-GROUP/hpx/issues/756>

⁵³¹³ <https://github.com/STELLAR-GROUP/hpx/issues/755>

⁵³¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/753>

⁵³¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/752>

⁵³¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/750>

⁵³¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/749>

⁵³¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/748>

⁵³¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/747>

⁵³²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/745>

⁵³²¹ <https://github.com/STELLAR-GROUP/hpx/issues/744>

⁵³²² <https://github.com/STELLAR-GROUP/hpx/issues/743>

⁵³²³ <https://github.com/STELLAR-GROUP/hpx/issues/742>

⁵³²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/741>

⁵³²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/739>

⁵³²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/738>

⁵³²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/737>

⁵³²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/736>

- Issue #735⁵³²⁹ - Create native ibverbs parcelport based on rdma operations
- Issue #734⁵³³⁰ - Threads stolen performance counter total is incorrect
- Issue #733⁵³³¹ - Test benchmarks need to be checked and fixed
- Issue #732⁵³³² - Build fails with Mac, using mac ports clang-3.3 on latest git branch
- Issue #731⁵³³³ - Add global start/stop API for performance counters
- Issue #730⁵³³⁴ - Performance counter values are apparently incorrect
- Issue #729⁵³³⁵ - Unhandled switch
- Issue #728⁵³³⁶ - Serialization of hpx::util::function between two localities causes seg faults
- Issue #727⁵³³⁷ - Memory counters on Mac OS X
- Issue #725⁵³³⁸ - Restore original thread priority on resume
- Issue #724⁵³³⁹ - Performance benchmarks do not depend on main HPX libraries
- Issue #723⁵³⁴⁰ - [teletype]–hpx:nodes=``cat \$PBS_NODEFILE`` works; –hpx:nodefile=\$PBS_NODEFILE does not.[c++]
- Issue #722⁵³⁴¹ - Fix binding const member functions as actions
- Issue #719⁵³⁴² - Create performance counter exposing compression ratio
- Issue #718⁵³⁴³ - Add possibility to compress parcel data
- Issue #717⁵³⁴⁴ - strip_credit_from_gid has misleading semantics
- Issue #716⁵³⁴⁵ - Non-option arguments to programs run using pbsdsh must be before --hpx:nodes, contrary to directions
- Issue #715⁵³⁴⁶ - Re-thrown exceptions should retain the original call site
- Issue #714⁵³⁴⁷ - failed assertion in debug mode
- Issue #713⁵³⁴⁸ - Add performance counters monitoring connection caches
- Issue #712⁵³⁴⁹ - Adjust parcel related performance counters to be connection type specific
- Issue #711⁵³⁵⁰ - configuration failure

⁵³²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/735>

⁵³³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/734>

⁵³³¹ <https://github.com/STELLAR-GROUP/hpx/issues/733>

⁵³³² <https://github.com/STELLAR-GROUP/hpx/issues/732>

⁵³³³ <https://github.com/STELLAR-GROUP/hpx/issues/731>

⁵³³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/730>

⁵³³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/729>

⁵³³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/728>

⁵³³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/727>

⁵³³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/725>

⁵³³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/724>

⁵³⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/723>

⁵³⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/722>

⁵³⁴² <https://github.com/STELLAR-GROUP/hpx/issues/719>

⁵³⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/718>

⁵³⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/717>

⁵³⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/716>

⁵³⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/715>

⁵³⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/714>

⁵³⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/713>

⁵³⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/712>

⁵³⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/711>

- Issue #710⁵³⁵¹ - Error “timed out while trying to find room in the connection cache” when trying to start multiple localities on a single computer
- Issue #709⁵³⁵² - Add new thread state ‘staged’ referring to task descriptions
- Issue #708⁵³⁵³ - Detect/mitigate bad non-system installs of GCC on Redhat systems
- Issue #707⁵³⁵⁴ - Many examples do not link with Git HEAD version
- Issue #706⁵³⁵⁵ - `hpx::init` removes portions of non-option command line arguments before last = sign
- Issue #705⁵³⁵⁶ - Create rolling average and median aggregating performance counters
- Issue #704⁵³⁵⁷ - Create performance counter to expose thread queue waiting time
- Issue #703⁵³⁵⁸ - Add support to HPX build system to find `libcrtool.a` and related headers
- Issue #699⁵³⁵⁹ - Generalize instrumentation support
- Issue #698⁵³⁶⁰ - compilation failure with `hwloc` absent
- Issue #697⁵³⁶¹ - Performance counter counts should be zero indexed
- Issue #696⁵³⁶² - Distributed problem
- Issue #695⁵³⁶³ - Bad perf counter time printed
- Issue #693⁵³⁶⁴ - --help doesn't print component specific command line options
- Issue #692⁵³⁶⁵ - SLURM support broken
- Issue #691⁵³⁶⁶ - exception while executing any application linked with `hwloc`
- Issue #690⁵³⁶⁷ - `thread_id_test` and `thread_launcher_test` failing
- Issue #689⁵³⁶⁸ - Make the buildbots use `hwloc`
- Issue #687⁵³⁶⁹ - compilation error fix (`hwloc_topology`)
- Issue #686⁵³⁷⁰ - Linker Error for Applications
- Issue #684⁵³⁷¹ - Pinning of service thread fails when number of worker threads equals the number of cores
- Issue #682⁵³⁷² - Add performance counters exposing number of stolen threads
- Issue #681⁵³⁷³ - Add `apply_continue` for asynchronous chaining of actions

⁵³⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/710>

⁵³⁵² <https://github.com/STELLAR-GROUP/hpx/issues/709>

⁵³⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/708>

⁵³⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/707>

⁵³⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/706>

⁵³⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/705>

⁵³⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/704>

⁵³⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/703>

⁵³⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/699>

⁵³⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/698>

⁵³⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/697>

⁵³⁶² <https://github.com/STELLAR-GROUP/hpx/issues/696>

⁵³⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/695>

⁵³⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/693>

⁵³⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/692>

⁵³⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/691>

⁵³⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/690>

⁵³⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/689>

⁵³⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/687>

⁵³⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/686>

⁵³⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/684>

⁵³⁷² <https://github.com/STELLAR-GROUP/hpx/issues/682>

⁵³⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/681>

- Issue #679⁵³⁷⁴ - Remove obsolete async_callback API functions
- Issue #678⁵³⁷⁵ - Add new API for setting/triggering LCOs
- Issue #677⁵³⁷⁶ - Add async_continue for true continuation style actions
- Issue #676⁵³⁷⁷ - Buildbot for gcc 4.4 broken
- Issue #675⁵³⁷⁸ - Partial preprocessing broken
- Issue #674⁵³⁷⁹ - HPX segfaults when built with gcc 4.7
- Issue #673⁵³⁸⁰ - use_guard_pages has inconsistent preprocessor guards
- Issue #672⁵³⁸¹ - External build breaks if library path has spaces
- Issue #671⁵³⁸² - release tarballs are tarbombs
- Issue #670⁵³⁸³ - CMake won't find Boost headers in layout=versioned install
- Issue #669⁵³⁸⁴ - Links in docs to source files broken if not installed
- Issue #667⁵³⁸⁵ - Not reading ini file properly
- Issue #664⁵³⁸⁶ - Adapt new meanings of ‘const’ and ‘mutable’
- Issue #661⁵³⁸⁷ - Implement BTL Parcel port
- Issue #655⁵³⁸⁸ - Make HPX work with the “decltype” result_of
- Issue #647⁵³⁸⁹ - documentation for specifying the number of high priority threads --hpx:high-priority-threads
- Issue #643⁵³⁹⁰ - Error parsing host file
- Issue #642⁵³⁹¹ - HWLoc issue with TAU
- Issue #639⁵³⁹² - Logging potentially suspends a running thread
- Issue #634⁵³⁹³ - Improve error reporting from parcel layer
- Issue #627⁵³⁹⁴ - Add tests for async and apply overloads that accept regular C++ functions
- Issue #626⁵³⁹⁵ - hpx/future.hpp header
- Issue #601⁵³⁹⁶ - Intel support

⁵³⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/679>

⁵³⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/678>

⁵³⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/677>

⁵³⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/676>

⁵³⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/675>

⁵³⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/674>

⁵³⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/673>

⁵³⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/672>

⁵³⁸² <https://github.com/STELLAR-GROUP/hpx/issues/671>

⁵³⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/670>

⁵³⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/669>

⁵³⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/667>

⁵³⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/664>

⁵³⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/661>

⁵³⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/655>

⁵³⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/647>

⁵³⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/643>

⁵³⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/642>

⁵³⁹² <https://github.com/STELLAR-GROUP/hpx/issues/639>

⁵³⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/634>

⁵³⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/627>

⁵³⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/626>

⁵³⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/601>

- Issue #557⁵³⁹⁷ - Remove action codes
- Issue #531⁵³⁹⁸ - AGAS request and response classes should use switch statements
- Issue #529⁵³⁹⁹ - Investigate the state of hwloc support
- Issue #526⁵⁴⁰⁰ - Make HPX aware of hyper-threading
- Issue #518⁵⁴⁰¹ - Create facilities allowing to use plain arrays as action arguments
- Issue #473⁵⁴⁰² - hwloc thread binding is broken on CPUs with hyperthreading
- Issue #383⁵⁴⁰³ - Change result type detection for hpx::util::bind to use result_of protocol
- Issue #341⁵⁴⁰⁴ - Consolidate route code
- Issue #219⁵⁴⁰⁵ - Only copy arguments into actions once
- Issue #177⁵⁴⁰⁶ - Implement distributed AGAS
- Issue #43⁵⁴⁰⁷ - Support for Darwin (Xcode + Clang)

2.10.23 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 Jähkälä 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.

⁵³⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/557>

⁵³⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/531>

⁵³⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/529>

⁵⁴⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/526>

⁵⁴⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/518>

⁵⁴⁰² <https://github.com/STELLAR-GROUP/hpx/issues/473>

⁵⁴⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/383>

⁵⁴⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/341>

⁵⁴⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/219>

⁵⁴⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/177>

⁵⁴⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/43>

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 [Github⁵⁴⁰⁸](#). 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 License⁵⁴⁰⁹](#)).

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 #666⁵⁴¹⁰ - Segfault on calling hpx::finalize twice
- Issue #665⁵⁴¹¹ - Adding declaration num_of_cores
- Issue #662⁵⁴¹² - pkgconfig is building wrong
- Issue #660⁵⁴¹³ - Need uninterrupt function
- Issue #659⁵⁴¹⁴ - Move our logging library into a different namespace
- Issue #658⁵⁴¹⁵ - Dynamic performance counter types are broken
- Issue #657⁵⁴¹⁶ - HPX v0.9.5 (RC1) hello_world example segfaulting
- Issue #656⁵⁴¹⁷ - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- Issue #654⁵⁴¹⁸ - Integrate the Boost auto_index tool with documentation
- Issue #653⁵⁴¹⁹ - Make HPX build on OS X + Clang + libc++
- Issue #651⁵⁴²⁰ - Add fine-grained control for thread pinning
- Issue #650⁵⁴²¹ - Command line no error message when using -hpx:(anything)
- Issue #645⁵⁴²² - Command line aliases don't work in [teletype]``@file``[c++]
- Issue #644⁵⁴²³ - Terminated threads are not always properly cleaned up
- Issue #640⁵⁴²⁴ - future_data<T>::set_on_completed_ used without locks
- Issue #638⁵⁴²⁵ - hpx build with intel compilers fails on linux
- Issue #637⁵⁴²⁶ - --copy-dt-needed-entries breaks with gold

⁵⁴⁰⁸ <https://github.com/STELLAR-GROUP/hpx/>

⁵⁴⁰⁹ https://www.boost.org/LICENSE_1_0.txt

⁵⁴¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/666>

⁵⁴¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/665>

5412 <https://github.com/STELLAR-GROUP/hpx/issues/662>5413 <https://github.com/STELLAR-GROUP/hpx/issues/660>5414 <https://github.com/STELLAR-GROUP/hpx/issues/659>5415 <https://github.com/STELLAR-GROUP/hpx/issues/658>5416 <https://github.com/STELLAR-GROUP/hpx/issues/657>5417 <https://github.com/STELLAR-GROUP/hpx/issues/656>5418 <https://github.com/STELLAR-GROUP/hpx/issues/654>5419 <https://github.com/STELLAR-GROUP/hpx/issues/653>5420 <https://github.com/STELLAR-GROUP/hpx/issues/651>5421 <https://github.com/STELLAR-GROUP/hpx/issues/650>5422 <https://github.com/STELLAR-GROUP/hpx/issues/645>5423 <https://github.com/STELLAR-GROUP/hpx/issues/644>5424 <https://github.com/STELLAR-GROUP/hpx/issues/640>5425 <https://github.com/STELLAR-GROUP/hpx/issues/638>5426 <https://github.com/STELLAR-GROUP/hpx/issues/637>

- Issue #635⁵⁴²⁷ - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- Issue #633⁵⁴²⁸ - Re-add examples to final 0.9.5 release
- Issue #632⁵⁴²⁹ - Example `thread_aware_timer` is broken
- Issue #631⁵⁴³⁰ - FFT application throws error in parcellayer
- Issue #630⁵⁴³¹ - Event synchronization example is broken
- Issue #629⁵⁴³² - Waiting on futures hangs
- Issue #628⁵⁴³³ - Add an `HPX_ALWAYS_ASSERT` macro
- Issue #625⁵⁴³⁴ - Port coroutines context switch benchmark
- Issue #621⁵⁴³⁵ - New INI section for stack sizes
- Issue #618⁵⁴³⁶ - `pkg_config` support does not work with a HPX debug build
- Issue #617⁵⁴³⁷ - `hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: 'get_thread_id'` was not declared in this scope
- Issue #616⁵⁴³⁸ - Change `wait_xxx` not to use locking
- Issue #615⁵⁴³⁹ - Revert visibility ‘fix’ (`fb0b6b8245dad1127b0c25ebaf9386b3945cca9`)
- Issue #614⁵⁴⁴⁰ - Fix Dataflow linker error
- Issue #613⁵⁴⁴¹ - `find_here` should throw an exception on failure
- Issue #612⁵⁴⁴² - Thread phase doesn’t show up in debug mode
- Issue #611⁵⁴⁴³ - Make stack guard pages configurable at runtime (initialization time)
- Issue #610⁵⁴⁴⁴ - Co-Locate Components
- Issue #609⁵⁴⁴⁵ - `future_overhead`
- Issue #608⁵⁴⁴⁶ - `--hpx:list-counter-infos` problem
- Issue #607⁵⁴⁴⁷ - Update Boost.Context based backend for coroutines
- Issue #606⁵⁴⁴⁸ - `1d_wave_equation` is not working
- Issue #605⁵⁴⁴⁹ - Any C++ function that has serializable arguments and a serializable return type should be re-

⁵⁴²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/635>

⁵⁴²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/633>

⁵⁴²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/632>

⁵⁴³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/631>

⁵⁴³¹ <https://github.com/STELLAR-GROUP/hpx/issues/630>

⁵⁴³² <https://github.com/STELLAR-GROUP/hpx/issues/629>

⁵⁴³³ <https://github.com/STELLAR-GROUP/hpx/issues/628>

⁵⁴³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/625>

⁵⁴³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/621>

⁵⁴³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/618>

⁵⁴³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/617>

⁵⁴³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/616>

⁵⁴³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/615>

⁵⁴⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/614>

⁵⁴⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/613>

⁵⁴⁴² <https://github.com/STELLAR-GROUP/hpx/issues/612>

⁵⁴⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/611>

⁵⁴⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/610>

⁵⁴⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/609>

⁵⁴⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/608>

⁵⁴⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/607>

⁵⁴⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/606>

⁵⁴⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/605>

mutable

- Issue #604⁵⁴⁵⁰ - Connecting localities isn't working anymore
- Issue #603⁵⁴⁵¹ - Do not verify any ini entries read from a file
- Issue #602⁵⁴⁵² - Rename argument_size to type_size/ added implementation to get parcel size
- Issue #599⁵⁴⁵³ - Enable locality specific command line options
- Issue #598⁵⁴⁵⁴ - Need an API that accesses the performance counter reporting the system uptime
- Issue #597⁵⁴⁵⁵ - compiling on ranger
- Issue #595⁵⁴⁵⁶ - I need a place to store data in a thread self pointer
- Issue #594⁵⁴⁵⁷ - 32/64 interoperability
- Issue #593⁵⁴⁵⁸ - Warn if logging is disabled at compile time but requested at runtime
- Issue #592⁵⁴⁵⁹ - Add optional argument value to --hpx:list-counters and --hpx:list-counter-infos
- Issue #591⁵⁴⁶⁰ - Allow for wildcards in performance counter names specified with --hpx:print-counter
- Issue #590⁵⁴⁶¹ - Local promise semantic differences
- Issue #589⁵⁴⁶² - Create API to query performance counter names
- Issue #587⁵⁴⁶³ - Add get_num_localities and get_num_threads to AGAS API
- Issue #586⁵⁴⁶⁴ - Adjust local AGAS cache size based on number of localities
- Issue #585⁵⁴⁶⁵ - Error while using counters in HPX
- Issue #584⁵⁴⁶⁶ - counting argument size of actions, initial pass.
- Issue #581⁵⁴⁶⁷ - Remove RemoteResult template parameter for future<>
- Issue #580⁵⁴⁶⁸ - Add possibility to hook into actions
- Issue #578⁵⁴⁶⁹ - Use angle brackets in HPX error dumps
- Issue #576⁵⁴⁷⁰ - Exception incorrectly thrown when --help is used
- Issue #575⁵⁴⁷¹ - HPX(bad_component_type) with gcc 4.7.2 and boost 1.51

⁵⁴⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/604>

⁵⁴⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/603>

⁵⁴⁵² <https://github.com/STELLAR-GROUP/hpx/issues/602>

⁵⁴⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/599>

⁵⁴⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/598>

⁵⁴⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/597>

⁵⁴⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/595>

⁵⁴⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/594>

⁵⁴⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/593>

⁵⁴⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/592>

⁵⁴⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/591>

⁵⁴⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/590>

⁵⁴⁶² <https://github.com/STELLAR-GROUP/hpx/issues/589>

⁵⁴⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/587>

⁵⁴⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/586>

⁵⁴⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/585>

⁵⁴⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/584>

⁵⁴⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/581>

⁵⁴⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/580>

⁵⁴⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/578>

⁵⁴⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/576>

⁵⁴⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/575>

- Issue #574⁵⁴⁷² - --hpx:connect command line parameter not working correctly
- Issue #571⁵⁴⁷³ - hpx::wait() (callback version) should pass the future to the callback function
- Issue #570⁵⁴⁷⁴ - hpx::wait should operate on boost::arrays and std::lists
- Issue #569⁵⁴⁷⁵ - Add a logging sink for Android
- Issue #568⁵⁴⁷⁶ - 2-argument version of HPX_DEFINE_COMPONENT_ACTION
- Issue #567⁵⁴⁷⁷ - Connecting to a running HPX application works only once
- Issue #565⁵⁴⁷⁸ - HPX doesn't shutdown properly
- Issue #564⁵⁴⁷⁹ - Partial preprocessing of new component creation interface
- Issue #563⁵⁴⁸⁰ - Add hpx::start/hpx::stop to avoid blocking main thread
- Issue #562⁵⁴⁸¹ - All command line arguments swallowed by hpx
- Issue #561⁵⁴⁸² - Boost.Tuple is not move aware
- Issue #558⁵⁴⁸³ - boost::shared_ptr<> style semantics/syntax for client classes
- Issue #556⁵⁴⁸⁴ - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
- Issue #555⁵⁴⁸⁵ - BOOST_FORCEINLINE does not name a type
- Issue #554⁵⁴⁸⁶ - Possible race condition in thread get_id()
- Issue #552⁵⁴⁸⁷ - Move enable client_base
- Issue #550⁵⁴⁸⁸ - Add stack size category 'huge'
- Issue #549⁵⁴⁸⁹ - ShenEOS run seg-faults on single or distributed runs
- Issue #545⁵⁴⁹⁰ - AUTOGLOB broken for add_hpx_component
- Issue #542⁵⁴⁹¹ - FindHPX_HDF5 still searches multiple times
- Issue #541⁵⁴⁹² - Quotes around application name in hpx::init
- Issue #539⁵⁴⁹³ - Race condition occurring with new lightweight threads
- Issue #535⁵⁴⁹⁴ - hpx_run_tests.py exits with no error code when tests are missing

⁵⁴⁷² <https://github.com/STELLAR-GROUP/hpx/issues/574>

⁵⁴⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/571>

⁵⁴⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/570>

⁵⁴⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/569>

⁵⁴⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/568>

⁵⁴⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/567>

⁵⁴⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/565>

⁵⁴⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/564>

⁵⁴⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/563>

⁵⁴⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/562>

⁵⁴⁸² <https://github.com/STELLAR-GROUP/hpx/issues/561>

⁵⁴⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/558>

⁵⁴⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/556>

⁵⁴⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/555>

⁵⁴⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/554>

⁵⁴⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/552>

⁵⁴⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/550>

⁵⁴⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/549>

⁵⁴⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/545>

⁵⁴⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/542>

⁵⁴⁹² <https://github.com/STELLAR-GROUP/hpx/issues/541>

⁵⁴⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/539>

⁵⁴⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/535>

- Issue #530⁵⁴⁹⁵ - Thread description(<unknown>) in logs
- Issue #523⁵⁴⁹⁶ - Make thread objects more lightweight
- Issue #521⁵⁴⁹⁷ - `hpx::error_code` is not usable for lightweight error handling
- Issue #520⁵⁴⁹⁸ - Add full user environment to HPX logs
- Issue #519⁵⁴⁹⁹ - Build succeeds, running fails
- Issue #517⁵⁵⁰⁰ - Add a guard page to linux coroutine stacks
- Issue #516⁵⁵⁰¹ - `hpx::thread::detach` suspends while holding locks, leads to hang in debug
- Issue #514⁵⁵⁰² - Preprocessed headers for <hpx/apply.hpp> don't compile
- Issue #513⁵⁵⁰³ - Buildbot configuration problem
- Issue #512⁵⁵⁰⁴ - Implement action based stack size customization
- Issue #511⁵⁵⁰⁵ - Move action priority into a separate type trait
- Issue #510⁵⁵⁰⁶ - trunk broken
- Issue #507⁵⁵⁰⁷ - no matching function for call to `boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(hpx::thre...`
- Issue #505⁵⁵⁰⁸ - undefined_symbol regression test currently failing
- Issue #502⁵⁵⁰⁹ - Adding OpenCL and OCLM support to HPX for Windows and Linux
- Issue #501⁵⁵¹⁰ - `find_package(HPX)` sets cmake output variables
- Issue #500⁵⁵¹¹ - `wait_any/wait_all` are badly named
- Issue #499⁵⁵¹² - Add support for disabling pbs support in pbs runs
- Issue #498⁵⁵¹³ - Error during no-cache runs
- Issue #496⁵⁵¹⁴ - Add partial preprocessing support to cmake
- Issue #495⁵⁵¹⁵ - Support HPX modules exporting startup/shutdown functions only
- Issue #494⁵⁵¹⁶ - Allow modules to specify when to run startup/shutdown functions
- Issue #493⁵⁵¹⁷ - Avoid constructing a string in `make_success_code`

5495 <https://github.com/STELLAR-GROUP/hpx/issues/530>

5496 <https://github.com/STELLAR-GROUP/hpx/issues/523>

5497 <https://github.com/STELLAR-GROUP/hpx/issues/521>

5498 <https://github.com/STELLAR-GROUP/hpx/issues/520>

5499 <https://github.com/STELLAR-GROUP/hpx/issues/519>

5500 <https://github.com/STELLAR-GROUP/hpx/issues/517>

5501 <https://github.com/STELLAR-GROUP/hpx/issues/516>

5502 <https://github.com/STELLAR-GROUP/hpx/issues/514>

5503 <https://github.com/STELLAR-GROUP/hpx/issues/513>

5504 <https://github.com/STELLAR-GROUP/hpx/issues/512>

5505 <https://github.com/STELLAR-GROUP/hpx/issues/511>

5506 <https://github.com/STELLAR-GROUP/hpx/issues/510>

5507 <https://github.com/STELLAR-GROUP/hpx/issues/507>

5508 <https://github.com/STELLAR-GROUP/hpx/issues/505>

5509 <https://github.com/STELLAR-GROUP/hpx/issues/502>

5510 <https://github.com/STELLAR-GROUP/hpx/issues/501>

5511 <https://github.com/STELLAR-GROUP/hpx/issues/500>

5512 <https://github.com/STELLAR-GROUP/hpx/issues/499>

5513 <https://github.com/STELLAR-GROUP/hpx/issues/498>

5514 <https://github.com/STELLAR-GROUP/hpx/issues/496>

5515 <https://github.com/STELLAR-GROUP/hpx/issues/495>

5516 <https://github.com/STELLAR-GROUP/hpx/issues/494>

5517 <https://github.com/STELLAR-GROUP/hpx/issues/493>

- Issue #492⁵⁵¹⁸ - Performance counter creation is no longer synchronized at startup
- Issue #491⁵⁵¹⁹ - Performance counter creation is no longer synchronized at startup
- Issue #490⁵⁵²⁰ - Sheneos on_completed_bulk seg fault in distributed
- Issue #489⁵⁵²¹ - compiling issue with g++44
- Issue #488⁵⁵²² - Adding OpenCL and OCLM support to HPX for the MSVC platform
- Issue #487⁵⁵²³ - FindHPX.cmake problems
- Issue #485⁵⁵²⁴ - Change distributing_factory and binpacking_factory to use bulk creation
- Issue #484⁵⁵²⁵ - Change HPX_DONT_USE_PREPROCESSED_FILES to HPX_USE_PREPROCESSED_FILES
- Issue #483⁵⁵²⁶ - Memory counter for Windows
- Issue #479⁵⁵²⁷ - strange errors appear when requesting performance counters on multiple nodes
- Issue #477⁵⁵²⁸ - Create (global) timer for multi-threaded measurements
- Issue #472⁵⁵²⁹ - Add partial preprocessing using Wave
- Issue #471⁵⁵³⁰ - Segfault stack traces don't show up in release
- Issue #468⁵⁵³¹ - External projects need to link with internal components
- Issue #462⁵⁵³² - Startup/shutdown functions are called more than once
- Issue #458⁵⁵³³ - Consolidate hpx::util::high_resolution_timer and hpx::util::high_resolution_clock
- Issue #457⁵⁵³⁴ - index out of bounds in allgather_and_gate on 4 cores or more
- Issue #448⁵⁵³⁵ - Make HPX compile with clang
- Issue #447⁵⁵³⁶ - ‘make tests’ should execute tests on local installation
- Issue #446⁵⁵³⁷ - Remove SVN-related code from the codebase
- Issue #444⁵⁵³⁸ - race condition in smp
- Issue #441⁵⁵³⁹ - Patched Boost.Serialization headers should only be installed if needed
- Issue #439⁵⁵⁴⁰ - Components using HPX_REGISTER_STARTUP_MODULE fail to compile with MSVC

⁵⁵¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/492>

⁵⁵¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/491>

⁵⁵²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/490>

⁵⁵²¹ <https://github.com/STELLAR-GROUP/hpx/issues/489>

⁵⁵²² <https://github.com/STELLAR-GROUP/hpx/issues/488>

⁵⁵²³ <https://github.com/STELLAR-GROUP/hpx/issues/487>

⁵⁵²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/485>

⁵⁵²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/484>

⁵⁵²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/483>

⁵⁵²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/479>

⁵⁵²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/477>

⁵⁵²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/472>

⁵⁵³⁰ <https://github.com/STELLAR-GROUP/hpx/issues/471>

⁵⁵³¹ <https://github.com/STELLAR-GROUP/hpx/issues/468>

⁵⁵³² <https://github.com/STELLAR-GROUP/hpx/issues/462>

⁵⁵³³ <https://github.com/STELLAR-GROUP/hpx/issues/458>

⁵⁵³⁴ <https://github.com/STELLAR-GROUP/hpx/issues/457>

⁵⁵³⁵ <https://github.com/STELLAR-GROUP/hpx/issues/448>

⁵⁵³⁶ <https://github.com/STELLAR-GROUP/hpx/issues/447>

⁵⁵³⁷ <https://github.com/STELLAR-GROUP/hpx/issues/446>

⁵⁵³⁸ <https://github.com/STELLAR-GROUP/hpx/issues/444>

⁵⁵³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/441>

⁵⁵⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/439>

- Issue #436⁵⁵⁴¹ - Verify that no locks are being held while threads are suspended
- Issue #435⁵⁵⁴² - Installing HPX should not clobber existing Boost installation
- Issue #434⁵⁵⁴³ - Logging external component failed (Boost 1.50)
- Issue #433⁵⁵⁴⁴ - Runtime crash when building all examples
- Issue #432⁵⁵⁴⁵ - Dataflow hangs on 512 cores/64 nodes
- Issue #430⁵⁵⁴⁶ - Problem with distributing factory
- Issue #424⁵⁵⁴⁷ - File paths referring to XSL-files need to be properly escaped
- Issue #417⁵⁵⁴⁸ - Make dataflow LCOs work out of the box by using partial preprocessing
- Issue #413⁵⁵⁴⁹ - hpx_svnversion.py fails on Windows
- Issue #412⁵⁵⁵⁰ - Make hpx::error_code equivalent to hpx::exception
- Issue #398⁵⁵⁵¹ - HPX clobbers out-of-tree application specific CMake variables (specifically CMAKE_BUILD_TYPE)
- Issue #394⁵⁵⁵² - Remove code generating random port numbers for network
- Issue #378⁵⁵⁵³ - ShenEOS scaling issues
- Issue #354⁵⁵⁵⁴ - Create a coroutines wrapper for Boost.Context
- Issue #349⁵⁵⁵⁵ - Commandline option --localities=N/-lN should be necessary only on AGAS locality
- Issue #334⁵⁵⁵⁶ - Add auto_index support to cmake based documentation toolchain
- Issue #318⁵⁵⁵⁷ - Network benchmarks
- Issue #317⁵⁵⁵⁸ - Implement network performance counters
- Issue #310⁵⁵⁵⁹ - Duplicate logging entries
- Issue #230⁵⁵⁶⁰ - Add compile time option to disable thread debugging info
- Issue #171⁵⁵⁶¹ - Add an INI option to turn off deadlock detection independently of logging
- Issue #170⁵⁵⁶² - OSHL internal counters are incorrect
- Issue #103⁵⁵⁶³ - Better diagnostics for multiple component/action registrations under the same name

⁵⁵⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/436>

⁵⁵⁴² <https://github.com/STELLAR-GROUP/hpx/issues/435>

⁵⁵⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/434>

⁵⁵⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/433>

⁵⁵⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/432>

⁵⁵⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/430>

⁵⁵⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/424>

⁵⁵⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/417>

⁵⁵⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/413>

⁵⁵⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/412>

⁵⁵⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/398>

⁵⁵⁵² <https://github.com/STELLAR-GROUP/hpx/issues/394>

⁵⁵⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/378>

⁵⁵⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/354>

⁵⁵⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/349>

⁵⁵⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/334>

⁵⁵⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/318>

⁵⁵⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/317>

⁵⁵⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/310>

⁵⁵⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/230>

⁵⁵⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/171>

⁵⁵⁶² <https://github.com/STELLAR-GROUP/hpx/issues/170>

⁵⁵⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/103>

- Issue #48⁵⁵⁶⁴ - Support for Darwin (Xcode + Clang)
- Issue #21⁵⁵⁶⁵ - Build fails with GCC 4.6

2.10.24 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 Standard⁵⁵⁶⁶.
- *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:

⁵⁵⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/48>

⁵⁵⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/21>

⁵⁵⁶⁶ <http://www.open-std.org/jtc1/sc22/wg21>

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

int hpx_main()
{
    return hpx::finalize();
}

int main()
{
    return hpx::init();
}
```

This removes the immediate dependency on the Boost.Program Options⁵⁵⁶⁷ 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 Options⁵⁵⁶⁸, 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`.

⁵⁵⁶⁷ https://www.boost.org/doc/html/program_options.html

⁵⁵⁶⁸ 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 #71⁵⁵⁶⁹ - GIDs that are not serialized via `handle_gid`<> should raise an exception
- Issue #105⁵⁵⁷⁰ - Allow for `hpx::util::functions` to be registered in the AGAS symbolic namespace
- Issue #107⁵⁵⁷¹ - Nasty threadmanger race condition (reproducible in `sheneos_test`)
- Issue #108⁵⁵⁷² - Add millisecond resolution to *HPX* logs on Linux
- Issue #110⁵⁵⁷³ - Shutdown hang in distributed with release build
- Issue #116⁵⁵⁷⁴ - Don't use TSS for the applier and runtime pointers
- Issue #162⁵⁵⁷⁵ - Move local synchronous execution shortcut from `hpx::function` to the applier
- Issue #172⁵⁵⁷⁶ - Cache sources in CMake and check if they change manually
- Issue #178⁵⁵⁷⁷ - Add an INI option to turn off ranged-based AGAS caching
- Issue #187⁵⁵⁷⁸ - Support for disabling performance counter deployment
- Issue #202⁵⁵⁷⁹ - Support for sending performance counter data to a specific file
- Issue #218⁵⁵⁸⁰ - boost.coroutines allows different stack sizes, but stack pool is unaware of this
- Issue #231⁵⁵⁸¹ - Implement movable `boost::bind`
- Issue #232⁵⁵⁸² - Implement movable `boost::function`
- Issue #236⁵⁵⁸³ - Allow binding `hpx::util::function` to actions
- Issue #239⁵⁵⁸⁴ - Replace `hpx::function` with `hpx::util::function`
- Issue #240⁵⁵⁸⁵ - Can't specify `RemoteResult` with `lcos::async`
- Issue #242⁵⁵⁸⁶ - `REGISTER_TEMPLATE` support for plain actions
- Issue #243⁵⁵⁸⁷ - `handle_gid`<> support for `hpx::util::function`
- Issue #245⁵⁵⁸⁸ - `*_c_cache` code throws an exception if the queried GID is not in the local cache
- Issue #246⁵⁵⁸⁹ - Undefined references in dataflow/adaptive1d example

⁵⁵⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/71>

⁵⁵⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/105>

⁵⁵⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/107>

⁵⁵⁷² <https://github.com/STELLAR-GROUP/hpx/issues/108>

⁵⁵⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/110>

⁵⁵⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/116>

⁵⁵⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/162>

⁵⁵⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/172>

⁵⁵⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/178>

⁵⁵⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/187>

⁵⁵⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/202>

⁵⁵⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/218>

⁵⁵⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/231>

⁵⁵⁸² <https://github.com/STELLAR-GROUP/hpx/issues/232>

⁵⁵⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/236>

⁵⁵⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/239>

⁵⁵⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/240>

⁵⁵⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/242>

⁵⁵⁸⁷ <https://github.com/STELLAR-GROUP/hpx/issues/243>

⁵⁵⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/245>

⁵⁵⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/246>

- Issue #252⁵⁵⁹⁰ - Problems configuring sheneos with CMake
- Issue #254⁵⁵⁹¹ - Lifetime of components doesn't end when client goes out of scope
- Issue #259⁵⁵⁹² - CMake does not detect that MSVC10 has lambdas
- Issue #260⁵⁵⁹³ - io_service_pool segfault
- Issue #261⁵⁵⁹⁴ - Late parcel executed outside of pxthread
- Issue #263⁵⁵⁹⁵ - Cannot select allocator with CMake
- Issue #264⁵⁵⁹⁶ - Fix allocator select
- Issue #267⁵⁵⁹⁷ - Runtime error for hello_world
- Issue #269⁵⁵⁹⁸ - pthread_affinity_np test fails to compile
- Issue #270⁵⁵⁹⁹ - Compiler noise due to -Wcast-qual
- Issue #275⁵⁶⁰⁰ - Problem with configuration tests/include paths on Gentoo
- Issue #325⁵⁶⁰¹ - Sheneos is 200-400 times slower than the fortran equivalent
- Issue #331⁵⁶⁰² - `hpx::init` and `hpx_main()` should not depend on `program_options`
- Issue #333⁵⁶⁰³ - Add doxygen support to CMake for doc toolchain
- Issue #340⁵⁶⁰⁴ - Performance counters for parcels
- Issue #346⁵⁶⁰⁵ - Component loading error when running hello_world in distributed on MSVC2010
- Issue #362⁵⁶⁰⁶ - Missing initializer error
- Issue #363⁵⁶⁰⁷ - Parcel port serialization error
- Issue #366⁵⁶⁰⁸ - Parcel buffering leads to types incompatible exception
- Issue #368⁵⁶⁰⁹ - Scalable alternative to rand() needed for *HPX*
- Issue #369⁵⁶¹⁰ - IB over IP is substantially slower than just using standard TCP/IP
- Issue #374⁵⁶¹¹ - `hpx::lcos::wait` should work with dataflows and arbitrary classes meeting the future interface
- Issue #375⁵⁶¹² - Conflicting/ambiguous overloads of `hpx::lcos::wait`

⁵⁵⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/252>

⁵⁵⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/254>

⁵⁵⁹² <https://github.com/STELLAR-GROUP/hpx/issues/259>

⁵⁵⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/260>

⁵⁵⁹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/261>

⁵⁵⁹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/263>

⁵⁵⁹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/264>

⁵⁵⁹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/267>

⁵⁵⁹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/269>

⁵⁵⁹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/270>

⁵⁶⁰⁰ <https://github.com/STELLAR-GROUP/hpx/issues/275>

⁵⁶⁰¹ <https://github.com/STELLAR-GROUP/hpx/issues/325>

⁵⁶⁰² <https://github.com/STELLAR-GROUP/hpx/issues/331>

⁵⁶⁰³ <https://github.com/STELLAR-GROUP/hpx/issues/333>

⁵⁶⁰⁴ <https://github.com/STELLAR-GROUP/hpx/issues/340>

⁵⁶⁰⁵ <https://github.com/STELLAR-GROUP/hpx/issues/346>

⁵⁶⁰⁶ <https://github.com/STELLAR-GROUP/hpx/issues/362>

⁵⁶⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/363>

⁵⁶⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/366>

⁵⁶⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/368>

⁵⁶¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/369>

⁵⁶¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/374>

⁵⁶¹² <https://github.com/STELLAR-GROUP/hpx/issues/375>

- Issue #376⁵⁶¹³ - Find_HPX.cmake should set CMake variable HPX_FOUND for out of tree builds
- Issue #377⁵⁶¹⁴ - ShenEOS interpolate bulk and interpolate_one_bulk are broken
- Issue #379⁵⁶¹⁵ - Add support for distributed runs under SLURM
- Issue #382⁵⁶¹⁶ - _Unwind_Word not declared in boost.backtrace
- Issue #387⁵⁶¹⁷ - Doxygen should look only at list of specified files
- Issue #388⁵⁶¹⁸ - Running make install on an out-of-tree application is broken
- Issue #391⁵⁶¹⁹ - Out-of-tree application segfaults when running in qsub
- Issue #392⁵⁶²⁰ - Remove HPX_NO_INSTALL option from cmake build system
- Issue #396⁵⁶²¹ - Pragma related warnings when compiling with older gcc versions
- Issue #399⁵⁶²² - Out of tree component build problems
- Issue #400⁵⁶²³ - Out of source builds on Windows: linker should not receive compiler flags
- Issue #401⁵⁶²⁴ - Out of source builds on Windows: components need to be linked with hpx_serialization
- Issue #404⁵⁶²⁵ - gfortran fails to link automatically when fortran files are present
- Issue #405⁵⁶²⁶ - Inability to specify linking order for external libraries
- Issue #406⁵⁶²⁷ - Adapt action limits such that dataflow applications work without additional defines
- Issue #415⁵⁶²⁸ - locality_results is not a member of hpx::components::server
- Issue #425⁵⁶²⁹ - Breaking changes to traits::*result wrt std::vector<id_type>
- Issue #426⁵⁶³⁰ - AUTOLOB needs to be updated to support fortran

2.10.25 HPX V0.8.1 (Apr 21, 2012)

This is a point release including important bug fixes for *HPX V0.8.0 (Mar 23, 2012)*.

⁵⁶¹³ <https://github.com/STELLAR-GROUP/hpx/issues/376>

⁵⁶¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/377>

⁵⁶¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/379>

⁵⁶¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/382>

⁵⁶¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/387>

⁵⁶¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/388>

⁵⁶¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/391>

⁵⁶²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/392>

⁵⁶²¹ <https://github.com/STELLAR-GROUP/hpx/issues/396>

⁵⁶²² <https://github.com/STELLAR-GROUP/hpx/issues/399>

⁵⁶²³ <https://github.com/STELLAR-GROUP/hpx/issues/400>

⁵⁶²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/401>

⁵⁶²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/404>

⁵⁶²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/405>

⁵⁶²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/406>

⁵⁶²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/415>

⁵⁶²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/425>

⁵⁶³⁰ <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 #295⁵⁶³¹ - Don't require install path to be known at compile time.
- Issue #371⁵⁶³² - Add hpx iostreams to standard build.
- Issue #384⁵⁶³³ - Fix compilation with GCC 4.7.
- Issue #390⁵⁶³⁴ - Remove keep_factory_alive startup call from ShenEOS; add shutdown call to H5close.
- Issue #393⁵⁶³⁵ - 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.

2.10.26 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`<`o`> and `hpx::lcos::lazy_future`<`o`> 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.

⁵⁶³¹ <https://github.com/STELLAR-GROUP/hpx/issues/295>

⁵⁶³² <https://github.com/STELLAR-GROUP/hpx/issues/371>

5633 <https://github.com/STELLAR-GROUP/hpx/issues/384>5634 <https://github.com/STELLAR-GROUP/hpx/issues/390>5635 <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 benchmark⁵⁶³⁶ (thanks to Matthew Anderson for contributing this application).
- GTC (Gyrokinetic Toroidal Code)⁵⁶³⁷: a skeleton for particle in cell type codes.
- Random Memory Access: an example demonstrating random memory accesses in a large array
- ShenEOS example⁵⁶³⁸, demonstrating partitioning of large read-only data structures and exposing an interpolation API.
- Sine performance counter demo.
- Accumulator examples demonstrating how to write and use *HPX* components.
- Quickstart examples (like `hello_world`, `fibonacci`, `quicksort`, `factorial`, etc.) demonstrating simple *HPX* concepts which introduce some of the concepts in *HPX*.
- Load balancing and work stealing demos.

⁵⁶³⁶ <http://www.graph500.org/>

⁵⁶³⁷ <http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/gtc/>

⁵⁶³⁸ <http://stellarcollapse.org/equationofstate>

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 #31⁵⁶³⁹ - Specialize handle_gid<> for examples and tests
- Issue #72⁵⁶⁴⁰ - Fix AGAS reference counting
- Issue #104⁵⁶⁴¹ - heartbeat throws an exception when derefing the performance counter it's watching
- Issue #111⁵⁶⁴² - throttle causes an exception on the target application
- Issue #142⁵⁶⁴³ - One failed component loading causes an unrelated component to fail
- Issue #165⁵⁶⁴⁴ - Remote exception propagation bug in AGAS reference counting test
- Issue #186⁵⁶⁴⁵ - Test credit exhaustion/splitting (e.g. prepare_gid and symbol NS)
- Issue #188⁵⁶⁴⁶ - Implement remaining AGAS reference counting test cases
- Issue #258⁵⁶⁴⁷ - No type checking of GIDs in stubs classes
- Issue #271⁵⁶⁴⁸ - Seg fault/shared pointer assertion in distributed code

⁵⁶³⁹ <https://github.com/STELLAR-GROUP/hpx/issues/31>

⁵⁶⁴⁰ <https://github.com/STELLAR-GROUP/hpx/issues/72>

⁵⁶⁴¹ <https://github.com/STELLAR-GROUP/hpx/issues/104>

⁵⁶⁴² <https://github.com/STELLAR-GROUP/hpx/issues/111>

⁵⁶⁴³ <https://github.com/STELLAR-GROUP/hpx/issues/142>

⁵⁶⁴⁴ <https://github.com/STELLAR-GROUP/hpx/issues/165>

⁵⁶⁴⁵ <https://github.com/STELLAR-GROUP/hpx/issues/186>

⁵⁶⁴⁶ <https://github.com/STELLAR-GROUP/hpx/issues/188>

⁵⁶⁴⁷ <https://github.com/STELLAR-GROUP/hpx/issues/258>

⁵⁶⁴⁸ <https://github.com/STELLAR-GROUP/hpx/issues/271>

- Issue #281⁵⁶⁴⁹ - CMake options need descriptive text
- Issue #283⁵⁶⁵⁰ - AGAS caching broken (gva_cache needs to be rewritten with ICL)
- Issue #285⁵⁶⁵¹ - HPX_INSTALL root directory not the same as CMAKE_INSTALL_PREFIX
- Issue #286⁵⁶⁵² - New segfault in dataflow applications
- Issue #289⁵⁶⁵³ - Exceptions should only be logged if not handled
- Issue #290⁵⁶⁵⁴ - c++11 tests failure
- Issue #293⁵⁶⁵⁵ - Build target for component libraries
- Issue #296⁵⁶⁵⁶ - Compilation error with Boost V1.49rc1
- Issue #298⁵⁶⁵⁷ - Illegal instructions on termination
- Issue #299⁵⁶⁵⁸ - gravity aborts with multiple threads
- Issue #301⁵⁶⁵⁹ - Build error with Boost trunk
- Issue #303⁵⁶⁶⁰ - Logging assertion failure in distributed runs
- Issue #304⁵⁶⁶¹ - Exception ‘what’ strings are lost when exceptions from decode_parcel are reported
- Issue #306⁵⁶⁶² - Performance counter user interface issues
- Issue #307⁵⁶⁶³ - Logging exception in distributed runs
- Issue #308⁵⁶⁶⁴ - Logging deadlocks in distributed
- Issue #309⁵⁶⁶⁵ - Reference counting test failures and exceptions
- Issue #311⁵⁶⁶⁶ - Merge AGAS remote_interface with the runtime_support object
- Issue #314⁵⁶⁶⁷ - Object tracking for id_types
- Issue #315⁵⁶⁶⁸ - Remove handle_gid and handle credit splitting in id_type serialization
- Issue #320⁵⁶⁶⁹ - applier::get_locality_id() should return an error value (or throw an exception)
- Issue #321⁵⁶⁷⁰ - Optimization for id_types which are never split should be restored
- Issue #322⁵⁶⁷¹ - Command line processing ignored with Boost 1.47.0

⁵⁶⁴⁹ <https://github.com/STELLAR-GROUP/hpx/issues/281>

⁵⁶⁵⁰ <https://github.com/STELLAR-GROUP/hpx/issues/283>

⁵⁶⁵¹ <https://github.com/STELLAR-GROUP/hpx/issues/285>

⁵⁶⁵² <https://github.com/STELLAR-GROUP/hpx/issues/286>

⁵⁶⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/289>

⁵⁶⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/290>

⁵⁶⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/293>

⁵⁶⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/296>

⁵⁶⁵⁷ <https://github.com/STELLAR-GROUP/hpx/issues/298>

⁵⁶⁵⁸ <https://github.com/STELLAR-GROUP/hpx/issues/299>

⁵⁶⁵⁹ <https://github.com/STELLAR-GROUP/hpx/issues/301>

⁵⁶⁶⁰ <https://github.com/STELLAR-GROUP/hpx/issues/303>

⁵⁶⁶¹ <https://github.com/STELLAR-GROUP/hpx/issues/304>

⁵⁶⁶² <https://github.com/STELLAR-GROUP/hpx/issues/306>

⁵⁶⁶³ <https://github.com/STELLAR-GROUP/hpx/issues/307>

⁵⁶⁶⁴ <https://github.com/STELLAR-GROUP/hpx/issues/308>

⁵⁶⁶⁵ <https://github.com/STELLAR-GROUP/hpx/issues/309>

⁵⁶⁶⁶ <https://github.com/STELLAR-GROUP/hpx/issues/311>

⁵⁶⁶⁷ <https://github.com/STELLAR-GROUP/hpx/issues/314>

⁵⁶⁶⁸ <https://github.com/STELLAR-GROUP/hpx/issues/315>

⁵⁶⁶⁹ <https://github.com/STELLAR-GROUP/hpx/issues/320>

⁵⁶⁷⁰ <https://github.com/STELLAR-GROUP/hpx/issues/321>

⁵⁶⁷¹ <https://github.com/STELLAR-GROUP/hpx/issues/322>

- Issue #323⁵⁶⁷² - Credit exhaustion causes object to stay alive
- Issue #324⁵⁶⁷³ - Duplicate exception messages
- Issue #326⁵⁶⁷⁴ - Integrate Quickbook with CMake
- Issue #329⁵⁶⁷⁵ - --help and --version should still work
- Issue #330⁵⁶⁷⁶ - Create pkg-config files
- Issue #337⁵⁶⁷⁷ - Improve usability of performance counter timestamps
- Issue #338⁵⁶⁷⁸ - Non-std exceptions deriving from std::exceptions in tfunc may be sliced
- Issue #339⁵⁶⁷⁹ - Decrease the number of send_pending_parcels threads
- Issue #343⁵⁶⁸⁰ - Dynamically setting the stack size doesn't work
- Issue #351⁵⁶⁸¹ - 'make install' does not update documents
- Issue #353⁵⁶⁸² - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs
- Issue #355⁵⁶⁸³ - 'make' doesn't do anything after correct configuration
- Issue #356⁵⁶⁸⁴ - Don't use hpx::util::static_ in topology code
- Issue #359⁵⁶⁸⁵ - Infinite recursion in hpx::tuple serialization
- Issue #361⁵⁶⁸⁶ - Add compile time option to disable logging completely
- Issue #364⁵⁶⁸⁷ - Installation seriously broken in r7443

2.10.27 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.).

⁵⁶⁷² <https://github.com/STELLAR-GROUP/hpx/issues/323>

⁵⁶⁷³ <https://github.com/STELLAR-GROUP/hpx/issues/324>

⁵⁶⁷⁴ <https://github.com/STELLAR-GROUP/hpx/issues/326>

⁵⁶⁷⁵ <https://github.com/STELLAR-GROUP/hpx/issues/329>

⁵⁶⁷⁶ <https://github.com/STELLAR-GROUP/hpx/issues/330>

⁵⁶⁷⁷ <https://github.com/STELLAR-GROUP/hpx/issues/337>

⁵⁶⁷⁸ <https://github.com/STELLAR-GROUP/hpx/issues/338>

⁵⁶⁷⁹ <https://github.com/STELLAR-GROUP/hpx/issues/339>

⁵⁶⁸⁰ <https://github.com/STELLAR-GROUP/hpx/issues/343>

⁵⁶⁸¹ <https://github.com/STELLAR-GROUP/hpx/issues/351>

⁵⁶⁸² <https://github.com/STELLAR-GROUP/hpx/issues/353>

⁵⁶⁸³ <https://github.com/STELLAR-GROUP/hpx/issues/355>

⁵⁶⁸⁴ <https://github.com/STELLAR-GROUP/hpx/issues/356>

⁵⁶⁸⁵ <https://github.com/STELLAR-GROUP/hpx/issues/359>

⁵⁶⁸⁶ <https://github.com/STELLAR-GROUP/hpx/issues/361>

⁵⁶⁸⁷ <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 #28⁵⁶⁸⁸ - Integrate Windows/Linux CMake code for *HPX* core
- Issue #32⁵⁶⁸⁹ - `hpx::cout()` should be `hpx::cout`
- Issue #33⁵⁶⁹⁰ - AGAS V2 legacy client does not properly handle `error_code`
- Issue #60⁵⁶⁹¹ - AGAS: allow for registerid to optionally take ownership of the gid
- Issue #62⁵⁶⁹² - adaptive1d compilation failure in Fusion
- Issue #64⁵⁶⁹³ - Parcel subsystem doesn't resolve domain names
- Issue #83⁵⁶⁹⁴ - No error handling if no console is available
- Issue #84⁵⁶⁹⁵ - No error handling if a hosted locality is treated as the bootstrap server
- Issue #90⁵⁶⁹⁶ - Add general commandline option `-N`
- Issue #91⁵⁶⁹⁷ - Add possibility to read command line arguments from file
- Issue #92⁵⁶⁹⁸ - Always log exceptions/errors to the log file
- Issue #93⁵⁶⁹⁹ - Log the command line/program name
- Issue #95⁵⁷⁰⁰ - Support for distributed launches
- Issue #97⁵⁷⁰¹ - Attempt to create a bad component type in AMR examples
- Issue #100⁵⁷⁰² - factorial and factorial_get examples trigger AGAS component type assertions
- Issue #101⁵⁷⁰³ - Segfault when `hpx::process::here()` is called in fibonacci2
- Issue #102⁵⁷⁰⁴ - unknown_component_address in int_object_semaphore_client
- Issue #114⁵⁷⁰⁵ - marduk raises assertion with default parameters
- Issue #115⁵⁷⁰⁶ - Logging messages for SMP runs (on the console) shouldn't be buffered

⁵⁶⁸⁸ <https://github.com/STELLAR-GROUP/hpx/issues/28>

⁵⁶⁸⁹ <https://github.com/STELLAR-GROUP/hpx/issues/32>

⁵⁶⁹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/33>

⁵⁶⁹¹ <https://github.com/STELLAR-GROUP/hpx/issues/60>

⁵⁶⁹² <https://github.com/STELLAR-GROUP/hpx/issues/62>

⁵⁶⁹³ <https://github.com/STELLAR-GROUP/hpx/issues/64>

5694 <https://github.com/STELLAR-GROUP/hpx/issues/83>5695 <https://github.com/STELLAR-GROUP/hpx/issues/84>5696 <https://github.com/STELLAR-GROUP/hpx/issues/90>5697 <https://github.com/STELLAR-GROUP/hpx/issues/91>5698 <https://github.com/STELLAR-GROUP/hpx/issues/92>5699 <https://github.com/STELLAR-GROUP/hpx/issues/93>5700 <https://github.com/STELLAR-GROUP/hpx/issues/95>5701 <https://github.com/STELLAR-GROUP/hpx/issues/97>5702 <https://github.com/STELLAR-GROUP/hpx/issues/100>5703 <https://github.com/STELLAR-GROUP/hpx/issues/101>5704 <https://github.com/STELLAR-GROUP/hpx/issues/102>5705 <https://github.com/STELLAR-GROUP/hpx/issues/114>5706 <https://github.com/STELLAR-GROUP/hpx/issues/115>

- Issue #119⁵⁷⁰⁷ - marduk linking strategy breaks other applications
- Issue #121⁵⁷⁰⁸ - pbsdsh problem
- Issue #123⁵⁷⁰⁹ - marduk, dataflow and adaptive1d fail to build
- Issue #124⁵⁷¹⁰ - Lower default preprocessing arity
- Issue #125⁵⁷¹¹ - Move hpx::detail::diagnostic_information out of the detail namespace
- Issue #126⁵⁷¹² - Test definitions for AGAS reference counting
- Issue #128⁵⁷¹³ - Add averaging performance counter
- Issue #129⁵⁷¹⁴ - Error with endian.hpp while building adaptive1d
- Issue #130⁵⁷¹⁵ - Bad initialization of performance counters
- Issue #131⁵⁷¹⁶ - Add global startup/shutdown functions to component modules
- Issue #132⁵⁷¹⁷ - Avoid using auto_ptr
- Issue #133⁵⁷¹⁸ - On Windows hpx.dll doesn't get installed
- Issue #134⁵⁷¹⁹ - HPX_LIBRARY does not reflect real library name (on Windows)
- Issue #135⁵⁷²⁰ - Add detection of unique_ptr to build system
- Issue #137⁵⁷²¹ - Add command line option allowing to repeatedly evaluate performance counters
- Issue #139⁵⁷²² - Logging is broken
- Issue #140⁵⁷²³ - CMake problem on windows
- Issue #141⁵⁷²⁴ - Move all non-component libraries into \$PREFIX/lib/hpx
- Issue #143⁵⁷²⁵ - adaptive1d throws an exception with the default command line options
- Issue #146⁵⁷²⁶ - Early exception handling is broken
- Issue #147⁵⁷²⁷ - Sheneos doesn't link on Linux
- Issue #149⁵⁷²⁸ - sheneos_test hangs
- Issue #154⁵⁷²⁹ - Compilation fails for r5661

⁵⁷⁰⁷ <https://github.com/STELLAR-GROUP/hpx/issues/119>

⁵⁷⁰⁸ <https://github.com/STELLAR-GROUP/hpx/issues/121>

⁵⁷⁰⁹ <https://github.com/STELLAR-GROUP/hpx/issues/123>

⁵⁷¹⁰ <https://github.com/STELLAR-GROUP/hpx/issues/124>

⁵⁷¹¹ <https://github.com/STELLAR-GROUP/hpx/issues/125>

⁵⁷¹² <https://github.com/STELLAR-GROUP/hpx/issues/126>

⁵⁷¹³ <https://github.com/STELLAR-GROUP/hpx/issues/128>

⁵⁷¹⁴ <https://github.com/STELLAR-GROUP/hpx/issues/129>

⁵⁷¹⁵ <https://github.com/STELLAR-GROUP/hpx/issues/130>

⁵⁷¹⁶ <https://github.com/STELLAR-GROUP/hpx/issues/131>

⁵⁷¹⁷ <https://github.com/STELLAR-GROUP/hpx/issues/132>

⁵⁷¹⁸ <https://github.com/STELLAR-GROUP/hpx/issues/133>

⁵⁷¹⁹ <https://github.com/STELLAR-GROUP/hpx/issues/134>

⁵⁷²⁰ <https://github.com/STELLAR-GROUP/hpx/issues/135>

⁵⁷²¹ <https://github.com/STELLAR-GROUP/hpx/issues/137>

⁵⁷²² <https://github.com/STELLAR-GROUP/hpx/issues/139>

⁵⁷²³ <https://github.com/STELLAR-GROUP/hpx/issues/140>

⁵⁷²⁴ <https://github.com/STELLAR-GROUP/hpx/issues/141>

⁵⁷²⁵ <https://github.com/STELLAR-GROUP/hpx/issues/143>

⁵⁷²⁶ <https://github.com/STELLAR-GROUP/hpx/issues/146>

⁵⁷²⁷ <https://github.com/STELLAR-GROUP/hpx/issues/147>

⁵⁷²⁸ <https://github.com/STELLAR-GROUP/hpx/issues/149>

⁵⁷²⁹ <https://github.com/STELLAR-GROUP/hpx/issues/154>

- Issue #155⁵⁷³⁰ - Sine performance counters example chokes on chrono headers
- Issue #156⁵⁷³¹ - Add build type to –version
- Issue #157⁵⁷³² - Extend AGAS caching to store gid ranges
- Issue #158⁵⁷³³ - r5691 doesn't compile
- Issue #160⁵⁷³⁴ - Re-add AGAS function for resolving a locality to its prefix
- Issue #168⁵⁷³⁵ - Managed components should be able to access their own GID
- Issue #169⁵⁷³⁶ - Rewrite AGAS future pool
- Issue #179⁵⁷³⁷ - Complete switch to request class for AGAS server interface
- Issue #182⁵⁷³⁸ - Sine performance counter is loaded by other examples
- Issue #185⁵⁷³⁹ - Write tests for symbol namespace reference counting
- Issue #191⁵⁷⁴⁰ - Assignment of read-only variable in point_geometry
- Issue #200⁵⁷⁴¹ - Seg faults when querying performance counters
- Issue #204⁵⁷⁴² - –ifnames and suffix stripping needs to be more generic
- Issue #205⁵⁷⁴³ - –list-* and –print-counter-* options do not work together and produce no warning
- Issue #207⁵⁷⁴⁴ - Implement decrement entry merging
- Issue #208⁵⁷⁴⁵ - Replace the spinlocks in AGAS with hpx::lcos::local_mutexes
- Issue #210⁵⁷⁴⁶ - Add an –ifprefix option
- Issue #214⁵⁷⁴⁷ - Performance test for PX-thread creation
- Issue #216⁵⁷⁴⁸ - VS2010 compilation
- Issue #222⁵⁷⁴⁹ - r6045 context_linux_x86.hpp
- Issue #223⁵⁷⁵⁰ - fibonacci hangs when changing the state of an active thread
- Issue #225⁵⁷⁵¹ - Active threads end up in the FEB wait queue
- Issue #226⁵⁷⁵² - VS Build Error for Accumulator Client

5730 <https://github.com/STELLAR-GROUP/hpx/issues/155>

5731 <https://github.com/STELLAR-GROUP/hpx/issues/156>

5732 <https://github.com/STELLAR-GROUP/hpx/issues/157>

5733 <https://github.com/STELLAR-GROUP/hpx/issues/158>

5734 <https://github.com/STELLAR-GROUP/hpx/issues/160>

5735 <https://github.com/STELLAR-GROUP/hpx/issues/168>

5736 <https://github.com/STELLAR-GROUP/hpx/issues/169>

5737 <https://github.com/STELLAR-GROUP/hpx/issues/179>

5738 <https://github.com/STELLAR-GROUP/hpx/issues/182>

5739 <https://github.com/STELLAR-GROUP/hpx/issues/185>

5740 <https://github.com/STELLAR-GROUP/hpx/issues/191>

5741 <https://github.com/STELLAR-GROUP/hpx/issues/200>

5742 <https://github.com/STELLAR-GROUP/hpx/issues/204>

5743 <https://github.com/STELLAR-GROUP/hpx/issues/205>

5744 <https://github.com/STELLAR-GROUP/hpx/issues/207>

5745 <https://github.com/STELLAR-GROUP/hpx/issues/208>

5746 <https://github.com/STELLAR-GROUP/hpx/issues/210>

5747 <https://github.com/STELLAR-GROUP/hpx/issues/214>

5748 <https://github.com/STELLAR-GROUP/hpx/issues/216>

5749 <https://github.com/STELLAR-GROUP/hpx/issues/222>

5750 <https://github.com/STELLAR-GROUP/hpx/issues/223>

5751 <https://github.com/STELLAR-GROUP/hpx/issues/225>

5752 <https://github.com/STELLAR-GROUP/hpx/issues/226>

- Issue #228⁵⁷⁵³ - Move all traits into namespace hpx::traits
- Issue #229⁵⁷⁵⁴ - Invalid initialization of reference in thread_init_data
- Issue #235⁵⁷⁵⁵ - Invalid GID in iostreams
- Issue #238⁵⁷⁵⁶ - Demangle type names for the default implementation of get_action_name
- Issue #241⁵⁷⁵⁷ - C++11 support breaks GCC 4.5
- Issue #247⁵⁷⁵⁸ - Reference to temporary with GCC 4.4
- Issue #248⁵⁷⁵⁹ - Seg fault at shutdown with GCC 4.4
- Issue #253⁵⁷⁶⁰ - Default component action registration kills compiler
- Issue #272⁵⁷⁶¹ - G++ unrecognized command line option
- Issue #273⁵⁷⁶² - quicksort example doesn't compile
- Issue #277⁵⁷⁶³ - 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*: ⁵⁷⁶⁴. Use the Zenodo entry for referring to the latest version of *HPX*: ⁵⁷⁶⁵. Entries for citing specific versions of *HPX* can also be found at ⁵⁷⁶⁶.

2.12 HPX users

A list of institutions and projects using *HPX* can be found on the *HPX* Users⁵⁷⁶⁷ 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)⁵⁷⁶⁸, a multi-disciplinary

⁵⁷⁵³ <https://github.com/STELLAR-GROUP/hpx/issues/228>

⁵⁷⁵⁴ <https://github.com/STELLAR-GROUP/hpx/issues/229>

⁵⁷⁵⁵ <https://github.com/STELLAR-GROUP/hpx/issues/235>

⁵⁷⁵⁶ <https://github.com/STELLAR-GROUP/hpx/issues/238>

5757 <https://github.com/STELLAR-GROUP/hpx/issues/241>5758 <https://github.com/STELLAR-GROUP/hpx/issues/247>5759 <https://github.com/STELLAR-GROUP/hpx/issues/248>5760 <https://github.com/STELLAR-GROUP/hpx/issues/253>5761 <https://github.com/STELLAR-GROUP/hpx/issues/272>5762 <https://github.com/STELLAR-GROUP/hpx/issues/273>5763 <https://github.com/STELLAR-GROUP/hpx/issues/277>5764 <https://joss.theoj.org/papers/022e5917b95517dff20cd3742ab95eca>5765 <https://doi.org/10.5281/zenodo.598202>5766 <https://doi.org/10.5281/zenodo.598202>5767 <https://hpx.stellar-group.org/hpx-users/>5768 <https://www.cct.lsu.edu>

research institute at Louisiana State University (LSU)⁵⁷⁶⁹. The ParalleX group was working to develop a new and experimental execution model for future high performance computing architectures. This model was christened ParalleX. The first implementations of ParalleX were crude, and many of those designs had to be discarded entirely. However, over time the team learned quite a bit about how to design a parallel, distributed runtime system which implements the concepts of ParalleX.

From the very beginning, this endeavour has been a group effort. In addition to a handful of interested researchers, there have always been graduate and undergraduate students participating in the discussions, design, and implementation of *HPX*. In 2011 we decided to formalize our collective research efforts by creating the STE||AR⁵⁷⁷⁰ 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||AR⁵⁷⁷¹ 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 Boost⁵⁷⁷² 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||AR⁵⁷⁷³ and ParalleX, see *People* and *Why HPX?*.

2.13.2 People

The STE||AR⁵⁷⁷⁴ 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||AR⁵⁷⁷⁵ 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.

⁵⁷⁶⁹ <https://www.lsu.edu>

⁵⁷⁷⁰ <https://stellar-group.org>

⁵⁷⁷¹ <https://stellar-group.org>

⁵⁷⁷² <https://www.boost.org/>

⁵⁷⁷³ <https://stellar-group.org>

⁵⁷⁷⁴ <https://stellar-group.org>

⁵⁷⁷⁵ <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 2.87: Contributors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT) ⁵⁷⁷⁶ , Louisiana State University (LSU) ⁵⁷⁷⁷	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture ⁵⁷⁷⁸ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ⁵⁷⁷⁹	thom.heller@gmail.com
Agustin Berge		agustinberge@gmail.com
Mikael Simberg	Swiss National Supercomputing Centre ⁵⁷⁸⁰	simbergm@scs.ch
John Biddiscombe	Swiss National Supercomputing Centre ⁵⁷⁸¹	biddisco@scs.ch
Anton Bikineev	Center for Computation and Technology (CCT) ⁵⁷⁸² , Louisiana State University (LSU) ⁵⁷⁸³	ant.bikineev@gmail.com
Martin Stumpf	Department of Computer Science 3 - Computer Architecture ⁵⁷⁸⁴ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ⁵⁷⁸⁵	martin.h.stumpf@gmail.com
Bryce Adelstein Lelbach		brycelelbach@gmail.com
Shuangyang Yang	Center for Computation and Technology (CCT) ⁵⁷⁸⁶ , Louisiana State University (LSU) ⁵⁷⁸⁷	syang16@cct.lsu.edu
Jeroen Habraken		vexocide@gmail.com
Steven Brandt	Center for Computation and Technology (CCT) ⁵⁷⁸⁸ , Louisiana State University (LSU) ⁵⁷⁸⁹	sbrandt@cct.lsu.edu
Antoine Tran Tan	Paris-Saclay University ⁵⁷⁹⁰ ,	antoine.trantan@universite-paris-saclay.fr
Adrian S. Lemoine	AMD ⁵⁷⁹¹	Adrian.Lemoine@amd.com
Maciej Brodowicz		maciekab@gmail.com
Giannis Gondidelis	Center for Computation and Technology (CCT) ⁵⁷⁹² , Louisiana State University (LSU) ⁵⁷⁹³	gonidelis@hotmail.com

Contributors to this document

Table 2.88: Documentation authors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT) ⁵⁷⁹⁴ , Louisiana State University (LSU) ⁵⁷⁹⁵	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture ⁵⁷⁹⁶ , Friedrich-Alexander University Erlangen-Nuremberg (FAU) ⁵⁷⁹⁷	thom.heller@gmail.com
Bryce Adelstein Lelbach		brycelelbach@gmail.com
Vinay C Amatya	Center for Computation and Technology (CCT) ⁵⁷⁹⁸ , Louisiana State University (LSU) ⁵⁷⁹⁹	vamatya@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT) ⁵⁸⁰⁰ , Louisiana State University (LSU) ⁵⁸⁰¹	sbrandt@cct.lsu.edu
Maciej Brodowicz		maciekab@gmail.com
Adrian S. Lemoine	AMD ⁵⁸⁰²	Adrian.Lemoine@amd.com
Rebecca Stobaugh		rstobaugh1@gmail.com
Dimitra Karatza	Faculty of Electrical Engineering, Mathematics & Computer Science ⁵⁸⁰³ , Delft University of Technology ⁵⁸⁰⁴	dimitra.karatza11@gmail.com
Bhumit Attarde		bhumitattarde2@gmail.com

⁵⁷⁷⁶ <https://www.cct.lsu.edu>⁵⁷⁷⁷ <https://www.lsu.edu>⁵⁷⁷⁸ <https://www3.cs.fau.de>⁵⁷⁷⁹ <https://www.fau.de>⁵⁷⁸⁰ <https://www.cscs.ch>⁵⁷⁸¹ <https://www.cscs.ch>⁵⁷⁸² <https://www.cct.lsu.edu>⁵⁷⁸³ <https://www.lsu.edu>⁵⁷⁸⁴ <https://www3.cs.fau.de>⁵⁷⁸⁵ <https://www.fau.de>⁵⁷⁸⁶ <https://www.cct.lsu.edu>⁵⁷⁸⁷ <https://www.lsu.edu>⁵⁷⁸⁸ <https://www.cct.lsu.edu>⁵⁷⁸⁹ <https://www.lsu.edu>⁵⁷⁹⁰ <https://www.universite-paris-saclay.fr/en>⁵⁷⁹¹ <https://www.amd.com/en>⁵⁷⁹² <https://www.cct.lsu.edu>⁵⁷⁹³ <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.

- Shreyas Atre, for contributing fixes to our implementation of senders/receivers.
- Alexander Neumann, for contributing fixes to the cmake build system.
- Dimitra Karatza, for her work on refactoring the documentation and providing a new user-friendly environment during and after Google Season of Docs 2021.
- 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 Centre⁵⁸⁰⁵), 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 Centre⁵⁸⁰⁶), for his tireless help cleaning up and maintaining *HPX*.

⁵⁷⁹⁴ <https://www.cct.lsu.edu>

⁵⁷⁹⁵ <https://www.lsu.edu>

⁵⁷⁹⁶ <https://www3.cs.fau.de>

⁵⁷⁹⁷ <https://www.fau.de>

⁵⁷⁹⁸ <https://www.cct.lsu.edu>

⁵⁷⁹⁹ <https://www.lsu.edu>

⁵⁸⁰⁰ <https://www.cct.lsu.edu>

⁵⁸⁰¹ <https://www.lsu.edu>

⁵⁸⁰² <https://www.amd.com/en>

⁵⁸⁰³ <https://www.tudelft.nl/en/eemcs>

⁵⁸⁰⁴ <https://www.tudelft.nl/en/>

⁵⁸⁰⁵ <https://www.csccs.ch>

⁵⁸⁰⁶ <https://www.csccs.ch>

- 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. Chris also contributed support for RISC-V architectures.
- Shoshana Jakobovits, for her work on the resource partitioner.
- Denis Blank, who re-wrote our unwrapped function to accept plain values arbitrary containers, and properly deal with nested futures.
- Ajai V. George, who implemented several of the parallel algorithms.
- Taeguk Kwon, who worked on implementing parallel algorithms as well as adapting the parallel algorithms to the Ranges TS.
- Zach Byerly ([Louisiana State University \(LSU\)](#)⁵⁸⁰⁷), 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 \(LSU\)](#)⁵⁸⁰⁸), 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 \(LSU\)](#)⁵⁸⁰⁹), 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 \(LSU\)](#)⁵⁸¹⁰), 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](#)⁵⁸¹¹) with *HPX*. He reported various problems and submitted several patches to fix issues allowing for a better integration with [LibGeoDecomp](#)⁵⁸¹².

⁵⁸⁰⁷ <https://www.lsu.edu>

⁵⁸⁰⁸ <https://www.lsu.edu>

⁵⁸⁰⁹ <https://www.lsu.edu>

⁵⁸¹⁰ <https://www.lsu.edu>

⁵⁸¹¹ <https://www.libgeodecomp.org/>

⁵⁸¹² <https://www.libgeodecomp.org/>

- 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 \(LSU\)](#)⁵⁸¹³), 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](#)⁵⁸¹⁴ C/C++ package manager.
- Vinay C Amatya ([Louisiana State University \(LSU\)](#)⁵⁸¹⁵), who contributed to the documentation and provided some of the *HPX* examples.
- Kevin Huck and Nick Chaimov ([University of Oregon](#)⁵⁸¹⁶), 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](#)⁵⁸¹⁷).
- 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 \(CCT\)](#)⁵⁸¹⁸) who contributed to *HPX* the implementation of several parallel algorithms (as proposed by [N4313](#)⁵⁸¹⁹).
- Anuj Sharma and Christopher Bross ([Department of Computer Science 3 - Computer Architecture](#)⁵⁸²⁰), who worked on *HPX* in the context of the [Google Summer of Code](#)⁵⁸²¹ program 2014.
- Martin Stumpf ([Department of Computer Science 3 - Computer Architecture](#)⁵⁸²²), who rebuilt our contiguous testing infrastructure (see the [HPX Buildbot Website](#)⁵⁸²³). Martin is also working on [HPXCL](#)⁵⁸²⁴ (mainly all work related to [OpenCL](#)⁵⁸²⁵) and implementing an *HPX* backend for [POCL](#)⁵⁸²⁶, a portable computing language solution based on [OpenCL](#)⁵⁸²⁷.
- Grant Mercer ([University of Nevada, Las Vegas](#)⁵⁸²⁸), who helped creating many of the parallel algorithms (as proposed by [N4313](#)⁵⁸²⁹).

⁵⁸¹³ <https://www.lsu.edu>⁵⁸¹⁴ <https://www.conan.io/>⁵⁸¹⁵ <https://www.lsu.edu>⁵⁸¹⁶ <https://uoregon.edu/>⁵⁸¹⁷ <https://github.com/STELLAR-GROUP/hpxcl/>⁵⁸¹⁸ <https://www.cct.lsu.edu>⁵⁸¹⁹ <http://wg21.link/n4313>⁵⁸²⁰ <https://www3.cs.fau.de>⁵⁸²¹ <https://developers.google.com/open-source/soc/>⁵⁸²² <https://www3.cs.fau.de>⁵⁸²³ <http://rostam.cct.lsu.edu/>⁵⁸²⁴ <https://github.com/STELLAR-GROUP/hpxcl/>⁵⁸²⁵ <https://www.khronos.org/opencl/>⁵⁸²⁶ <https://portablecl.org/>⁵⁸²⁷ <https://www.khronos.org/opencl/>⁵⁸²⁸ <https://www.unlv.edu>⁵⁸²⁹ <http://wg21.link/n4313>

- Damond Howard (Louisiana State University (LSU)⁵⁸³⁰), who works on HPXCL⁵⁸³¹ (mainly all work related to CUDA⁵⁸³²).
- Christoph Junghans (Los Alamos National Lab), who helped making our buildsystem more portable.
- Antoine Tran Tan (Laboratoire de Recherche en Informatique, Paris), who worked on integrating HPX as a backend for NT2⁵⁸³³. He also contributed an implementation of an API similar to Fortran co-arrays on top of HPX.
- John Biddiscombe (Swiss National Supercomputing Centre⁵⁸³⁴), who helped with the BlueGene/Q port of HPX, implemented the parallel sort algorithm, and made several other contributions.
- Erik Schnetter (Perimeter Institute for Theoretical Physics), who greatly helped to make HPX more robust by submitting a large amount of problem reports, feature requests, and made several direct contributions.
- Mathias Gaunard (Metascale), who contributed several patches to reduce compile time warnings generated while compiling HPX.
- Andreas Buhr, who helped with improving our documentation, especially by suggesting some fixes for inconsistencies.
- Patricia Grubel (New Mexico State University⁵⁸³⁵), 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

⁵⁸³⁰ <https://www.lsu.edu>

⁵⁸³¹ <https://github.com/STELLAR-GROUP/hpxcl/>

⁵⁸³² https://www.nvidia.com/object/cuda_home_new.html

⁵⁸³³ <https://www.numscale.com/nt2/>

⁵⁸³⁴ <https://www.cscs.ch>

⁵⁸³⁵ <https://www.nmsu.edu>

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 Acknowledgements⁵⁸³⁶ lists current and past funding sources for *HPX*. Special thanks to Google Summer of Code⁵⁸³⁷ and Google Season of Docs⁵⁸³⁸ for the continuous support they provide which helps us enhance both our code and our documentation.

⁵⁸³⁶ <https://hpx.stellar-group.org/funding-acknowledgements/>

⁵⁸³⁷ <https://developers.google.com/open-source/soc/>

⁵⁸³⁸ <https://developers.google.com/season-of-docs>

**CHAPTER
THREE**

INDEX

- genindex

INDEX

Symbols

--hpx:affinity
 command line option, 102
--hpx:agas
 command line option, 101
--hpx:app-config
 command line option, 103
--hpx:attach-debugger
 command line option, 104
--hpx:bind
 command line option, 102
--hpx:config
 command line option, 103
--hpx:connect
 command line option, 101
--hpx:console
 command line option, 101
--hpx:cores
 command line option, 102
--hpx:debug-agas-log
 command line option, 103
--hpx:debug-app-log
 command line option, 103
--hpx:debug-clp
 command line option, 104
--hpx:debug-hpx-log
 command line option, 103
--hpx:debug-parcel-log
 command line option, 103
--hpx:debug-timing-log
 command line option, 103
--hpx:dump-config
 command line option, 103
--hpx:dump-config-initial
 command line option, 103
--hpx:endnodes
 command line option, 101
--hpx:exit
 command line option, 103
--hpx:expect-connecting-localities
 command line option, 102
--hpx:force_ipv4
 command line option, 102
--hpx:help
 command line option, 101
--hpx:high-priority-threads
 command line option, 103
--hpx:hpx
 command line option, 101
--hpx:ifprefix
 command line option, 102
--hpx:ifsuffix
 command line option, 101
--hpx:iftransform
 command line option, 102
--hpx:ignore-batch-env
 command line option, 102
--hpx:info
 command line option, 101
--hpx:ini
 command line option, 103
--hpx:list-component-types
 command line option, 103
--hpx:list-counter-infos
 command line option, 104
--hpx:list-counters
 command line option, 104
--hpx:list-symbolic-names
 command line option, 103
--hpx:localities
 command line option, 102
--hpx:no-csv-header
 command line option, 104
--hpx:node
 command line option, 102
--hpx:nodefile
 command line option, 101
--hpx:nodes
 command line option, 101
--hpx:numa-sensitive
 command line option, 103
--hpx:options-file
 command line option, 101
--hpx:print-bind

```
    command line option, 102
--hpx:print-counter
    command line option, 104
--hpx:print-counter-at
    command line option, 104
--hpx:print-counter-destination
    command line option, 104
--hpx:print-counter-format
    command line option, 104
--hpx:print-counter-interval
    command line option, 104
--hpx:print-counter-reset
    command line option, 104
--hpx:print-counters-locally
    command line option, 104
--hpx:pu-offset
    command line option, 102
--hpx:pu-step
    command line option, 102
--hpx:queuing
    command line option, 102
--hpx:reset-counters
    command line option, 104
--hpx:run-agas-server
    command line option, 101
--hpx:run-agas-server-only
    command line option, 101
--hpx:run-hpx-main
    command line option, 101
--hpx:threads
    command line option, 102
--hpx:use-process-mask
    command line option, 102
--hpx:version
    command line option, 101
--hpx:worker
    command line option, 101
```

A

```
Action, 224
actions (C++ type), 1208
Active Global Address Space, 223
AGAS, 223
AMPLIFIER_ROOT:PATH
    command line option, 61
applier (C++ type), 1381
```

B

```
BOOST_ROOT:PATH
    command line option, 60
BREATHE_APIDOC_ROOT:PATH
    command line option, 1418
```

C

```
command line option
    --hpx:affinity, 102
    --hpx:agas, 101
    --hpx:app-config, 103
    --hpx:attach-debugger, 104
    --hpx:bind, 102
    --hpx:config, 103
    --hpx:connect, 101
    --hpx:console, 101
    --hpx:cores, 102
    --hpx:debug-agas-log, 103
    --hpx:debug-app-log, 103
    --hpx:debug-clp, 104
    --hpx:debug-hpx-log, 103
    --hpx:debug-parcel-log, 103
    --hpx:debug-timing-log, 103
    --hpx:dump-config, 103
    --hpx:dump-config-initial, 103
    --hpx:endnodes, 101
    --hpx:exit, 103
    --hpx:expect-connecting-localities, 102
    --hpx:force_ipv4, 102
    --hpx:help, 101
    --hpx:high-priority-threads, 103
    --hpx:hpx, 101
    --hpx:ifprefix, 102
    --hpx:ifsuffix, 101
    --hpx:iftransform, 102
    --hpx:ignore-batch-env, 102
    --hpx:info, 101
    --hpx:ini, 103
    --hpx:list-component-types, 103
    --hpx:list-counter-infos, 104
    --hpx:list-counters, 104
    --hpx:list-symbolic-names, 103
    --hpx:localities, 102
    --hpx:no-csv-header, 104
    --hpx:node, 102
    --hpx:nodename, 101
    --hpx:nodes, 101
    --hpx:numa-sensitive, 103
    --hpx:options-file, 101
    --hpx:print-bind, 102
    --hpx:print-counter, 104
    --hpx:print-counter-at, 104
    --hpx:print-counter-destination, 104
    --hpx:print-counter-format, 104
    --hpx:print-counter-interval, 104
    --hpx:print-counter-reset, 104
    --hpx:print-counters-locally, 104
    --hpx:pu-offset, 102
    --hpx:pu-step, 102
    --hpx:queuing, 102
```

```
--hpx:reset-counters, 104
--hpx:run-agas-server, 101
--hpx:run-agas-server-only, 101
--hpx:run-hpx-main, 101
--hpx:threads, 102
--hpx:use-process-mask, 102
--hpx:version, 101
--hpx:worker, 101
AMPLIFIER_ROOT:PATH, 61
BOOST_ROOT:PATH, 60
BREATHE_APIDOC_ROOT:PATH, 1418
DOXYGEN_ROOT:PATH, 1418
HDF5_ROOT:PATH, 61
HPX_COROUTINES_WITH_SWAP_CONTEXT_EMULATION:BOOL, 56
HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL, 60
HPX_DATASTRUCTURES_WITH_ADAPT_STD_VARIANT:BOOL, 60
HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY:BOOL, 60
HPX_ITERATOR_SUPPORT_WITH_BOOST_ITERATOR_TRAVERSAL_TAG:BOOL, 60
HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_EQUIVALENTS:BOOL, 60
HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS:BOOL, 60
HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION:BOOL, 60
HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL, 60
HPX_SERIALIZATION_WITH_SUPPORTS_ENDIANESS:BOOL, 60
HPX_TOPOLOGY_WITH_ADDITIONAL_HWLOC_TESTING:BOOL, 60
HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL, 57
HPX_WITH_APEX, 43
HPX_WITH_APEX:BOOL, 58
HPX_WITH_ASIO_TAG:STRING, 54
HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL, 59
HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL, 51
HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH, 51
HPX_WITH_BUILD_BINARY_PACKAGE:BOOL, 51
HPX_WITH_CHECK_MODULE_DEPENDENCIES:BOOL, 51
HPX_WITH_COMPILE_ONLY_TESTS:BOOL, 54
HPX_WITH_COMPILER_WARNINGS:BOOL, 51
HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL, 51
HPX_WITH_COMPRESSION_BZIP2:BOOL, 51
HPX_WITH_COMPRESSION_SNAPPY:BOOL, 52
HPX_WITH_COMPRESSION_ZLIB:BOOL, 52
HPX_WITH_COROUTINE_COUNTERS:BOOL, 56
HPX_WITH_CUDA, 43
HPX_WITH_CUDA:BOOL, 52
HPX_WITH_CXX_STANDARD, 43
HPX_WITH_CXX_STANDARD:STRING, 52
HPX_WITH_DATAPAR:BOOL, 52
HPX_WITH_DATAPAR_BACKEND:STRING, 52
HPX_WITH_DATAPAR_VC_NO_LIBRARY:BOOL, 52
HPX_WITH_DEPRECATED_WARNINGS:BOOL, 52
HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL, 52
HPX_WITH_DISTRIBUTED_RUNTIME:BOOL, 54
HPX_WITH_DOCUMENTATION:BOOL, 54
HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS:STRING, 54
HPX_WITH_DYNAMIC_HPX_MAIN:BOOL, 54
HPX_WITH_EXAMPLES, 43
HPX_WITH_EXAMPLES:BOOL, 54
HPX_WITH_EXAMPLES_QTHREADS:BOOL, 54
HPX_WITH_EXECUTABLE_PREFIX:STRING, 54
HPX_WITH_FAULT_TOLERANCE:BOOL, 52
HPX_WITH_FETCH_LCI:BOOL, 55
HPX_WITH_FULL_RPATH:BOOL, 52
HPX_WITH_GCC_VERSION_CHECK:BOOL, 52
HPX_WITH_GENERIC_CONTEXT_COROUTINES, 43
HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL, 52
HPX_WITH_HIDDEN_VISIBILITY:BOOL, 52
HPX_WITH_HIP:BOOL, 52
HPX_WITH_IO_COUNTERS:BOOL, 55
HPX_WITH_IO_POOL:BOOL, 56
HPX_WITH_ITTNOTIFY:BOOL, 58
HPX_WITH_LCI_TAG:STRING, 55
HPX_WITH_MALLOC, 43
HPX_WITH_MALLOC:STRING, 52
HPX_WITH_MAX_CPU_COUNT, 43
HPX_WITH_MAX_CPU_COUNT:STRING, 56
HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING, 56
HPX_WITH_MODULES_AS_STATIC_LIBRARIES:BOOL, 52
HPX_WITH_NETWORKING:BOOL, 57
HPX_WITH_NICE_THREADLEVEL:BOOL, 53
HPX_WITH_PAPI:BOOL, 58
HPX_WITH_PARALLEL_TESTS_BIND_NONE:BOOL, 59
HPX_WITH_PARCEL_COALESCING:BOOL, 53
```

HPX_WITH_PARCEL_PROFILING:BOOL, 58
HPX_WITH_PARCELPORT_ACTION_COUNTERS:BOOL,
 57
HPX_WITH_PARCELPORT_COUNTERS:BOOL, 58
HPX_WITH_PARCELPORT_LCI:BOOL, 58
HPX_WITH_PARCELPORT_LIBFABRIC:BOOL, 58
HPX_WITH_PARCELPORT_MPI, 43
HPX_WITH_PARCELPORT_MPI:BOOL, 58
HPX_WITH_PARCELPORT_TCP, 43
HPX_WITH_PARCELPORT_TCP:BOOL, 58
HPX_WITH_PKGCONFIG:BOOL, 53
HPX_WITH_POWER_COUNTER:BOOL, 60
HPX_WITH_PRECOMPILED_HEADERS:BOOL, 53
HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL, 53
HPX_WITH_SANITIZERS:BOOL, 59
HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL, 56
HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL, Component, 224
 56
HPX_WITH_SPINLOCK_POOL_NUM:STRING, 56
HPX_WITH_STACKOVERFLOW_DETECTION:BOOL, 53
HPX_WITH_STACKTRACES:BOOL, 56
HPX_WITH_STACKTRACES_DEMANGLE_SYMBOLS:BOOL,
 56
HPX_WITH_STACKTRACES_STATIC_SYMBOLS:BOOL,
 56
HPX_WITH_STATIC_LINKING:BOOL, 53
HPX_WITH_TESTS, 43
HPX_WITH_TESTS:BOOL, 55
HPX_WITH_TESTS_BENCHMARKS:BOOL, 55
HPX_WITH_TESTS_DEBUG_LOG:BOOL, 59
HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING,
 59
HPX_WITH_TESTS_EXAMPLES:BOOL, 55
HPX_WITH_TESTS_EXTERNAL_BUILD:BOOL, 55
HPX_WITH_TESTS_HEADERS:BOOL, 55
HPX_WITH_TESTS_MAX_THREADS_PER_LOCALITY:STRING,
 59
HPX_WITH_TESTS_REGRESSIONS:BOOL, 55
HPX_WITH_TESTS_UNIT:BOOL, 55
HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING,
 56
HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL,
 56
HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL,
 56
HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL,
 57
HPX_WITH_THREAD_DEBUG_INFO:BOOL, 59
HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL, 59
HPX_WITH_THREAD_GUARD_PAGE:BOOL, 59
HPX_WITH_THREAD_IDLE_RATES:BOOL, 57
HPX_WITH_THREAD_LOCAL_STORAGE:BOOL, 57
HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL,
 57

HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL, 57
HPX_WITH_THREAD_STACK_MMAP:BOOL, 57
HPX_WITH_THREAD_STEALING_COUNTS:BOOL, 57
HPX_WITH_THREAD_TARGET_ADDRESS:BOOL, 57
HPX_WITH_TIMER_POOL:BOOL, 57
HPX_WITH_TOOLS:BOOL, 55
HPX_WITH_UNITY_BUILD:BOOL, 53
HPX_WITH_VALGRIND:BOOL, 59
HPX_WITH_VERIFY_LOCKS:BOOL, 59
HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL, 59
HPX_WITH_VIM_YCM:BOOL, 53
HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING,
 53
HWLOC_ROOT:PATH, 60
PAPI_ROOT:PATH, 61
SPHINX_ROOT:PATH, 1418

components (*C++ type*), 1300, 1312, 1368

D

DOXYGEN_ROOT:PATH
command line option, 1418

H

HDF5_ROOT:PATH
command line option, 61

hpx (*C++ type*), 283, 288, 289, 293, 295, 299, 304,
 307, 310, 312, 318, 323, 325, 337, 341, 351–
 353, 356, 359, 365, 368, 370, 373, 376, 379,
 383, 388, 397, 399, 401, 402, 404, 411, 417,
 419, 423, 427, 434, 437, 440, 444, 447, 449,
 453, 455, 457, 459, 461, 462, 464, 466, 468,
 472, 475, 480, 487, 490, 493, 495, 499, 501,
 505, 510, 514, 524, 531, 538, 541, 545, 550,
 555, 559, 579, 585, 595, 599, 603, 613, 619,
 622, 630, 635, 641, 651, 663, 669, 672, 676,
 680, 685, 698, 708, 716, 733, 738, 744, 754,
 761, 768, 775, 782, 785, 788, 792, 796, 800,
 803, 814, 820, 831, 845, 850, 853, 857, 862,
 866, 875, 878–881, 884–889, 891, 892, 895,
 897, 898, 901, 903, 905, 907, 910, 912, 913,
 920, 924, 926, 928, 929, 931–933, 935, 939,
 942, 944, 949, 958, 966, 970, 973, 977, 980,
 985–990, 992, 996–998, 1000–1002, 1004,
 1009, 1012, 1014–1016, 1018, 1021, 1022,
 1024–1027, 1029, 1032–1034, 1036, 1038–
 1040, 1042, 1043, 1050–1052, 1055, 1058,
 1061, 1062, 1064, 1065, 1067, 1072, 1074,
 1077–1079, 1081, 1087–1090, 1098, 1100–
 1102, 1104, 1105, 1108, 1109, 1111, 1113,
 1127, 1131, 1133, 1136, 1142, 1143, 1145,
 1146, 1148, 1154, 1155, 1158, 1160, 1166–
 1169, 1174, 1176, 1183, 1185, 1187, 1191,
 1195–1198, 1205, 1207, 1208, 1210, 1211,

1214, 1215, 1230, 1237, 1238, 1241, 1243, `hpx::agas::addressing_service::enable_refcnt_caching_`
 1245, 1248, 1250, 1255, 1263–1265, 1267, *(C++ member)*, 1229
 1268, 1271, 1272, 1275, 1277–1279, 1281, `hpx::agas::addressing_service::end_migration`
(C++ function), 1228
 1283, 1286, 1287, 1289, 1292, 1293, 1296,
 1299, 1300, 1303, 1306–1308, 1311, 1312, `hpx::agas::addressing_service::garbage_collect`
(C++ function), 1216
 1314, 1316, 1318, 1320, 1322, 1324, 1325, `hpx::agas::addressing_service::garbage_collect_non_blocking`
 1328, 1331, 1332, 1335–1338, 1340, 1348, *(C++ function)*, 1216
 1357, 1360, 1363, 1365–1368, 1370, 1374,
 1379, 1381–1383, 1385, 1388, 1389, 1391–
 1393, 1398, 1401–1409, 1411, 1412 `hpx::agas::addressing_service::get_cache_entries`
(C++ function), 1216
`hpx::actions` (*C++ type*), 1207, 1208, 1210, 1211, `hpx::agas::addressing_service::get_cache_entry`
(C++ function), 1227
`hpx::actions::basic_action` (*C++ struct*), 1210 `hpx::agas::addressing_service::get_cache_erase_entry_count`
`hpx::adjacent_difference` (*C++ function*), 290–292 *(C++ function)*, 1217
`hpx::adjacent_find` (*C++ function*), 293 `hpx::agas::addressing_service::get_cache_erase_entry_time`
`hpx::agas` (*C++ type*), 1215, 1230, 1303, 1391 *(C++ function)*, 1217
`hpx::agas::addressing_service` (*C++ struct*), `hpx::agas::addressing_service::get_cache_evictions`
(C++ function), 1216
`hpx::agas::addressing_service::~addressing_service` `hpx::agas::addressing_service::get_cache_get_entry_count`
(C++ function), 1215 *(C++ function)*, 1216
`hpx::agas::addressing_service::action_priority` `hpx::agas::addressing_service::get_cache_get_entry_time`
(C++ member), 1229 *(C++ function)*, 1217
`hpx::agas::addressing_service::addressing_service` `hpx::agas::addressing_service::get_cache_hits`
(C++ function), 1215 *(C++ function)*, 1216
`hpx::agas::addressing_service::adjust_local_cache` `hpx::agas::addressing_service::get_cache_insertion_entry_count`
(C++ function), 1215 *(C++ function)*, 1216
`hpx::agas::addressing_service::begin_migration` `hpx::agas::addressing_service::get_cache_insertions`
(C++ function), 1228 *(C++ function)*, 1216
`hpx::agas::addressing_service::bind_async` `hpx::agas::addressing_service::get_cache_misses`
(C++ function), 1220 *(C++ function)*, 1216
`hpx::agas::addressing_service::bind_local` `hpx::agas::addressing_service::get_cache_update_entry_count`
(C++ function), 1220 *(C++ function)*, 1216
`hpx::agas::addressing_service::bind_postproc` `hpx::agas::addressing_service::get_colocation_id_async`
(C++ function), 1230 *(C++ function)*, 1224
`hpx::agas::addressing_service::bind_range_async` `hpx::agas::addressing_service::get_component_id`
(C++ function), 1221 *(C++ function)*, 1218
`hpx::agas::addressing_service::bind_range_local` `hpx::agas::addressing_service::get_component_type_name`
(C++ function), 1220 *(C++ function)*, 1219
`hpx::agas::addressing_service::bootstrap` `hpx::agas::addressing_service::get_console_locality`
(C++ function), 1215 *(C++ function)*, 1217
`hpx::agas::addressing_service::caching_` `hpx::agas::addressing_service::get_id_range`
(C++ member), 1229 *(C++ function)*, 1219
`hpx::agas::addressing_service::clear_cache` `hpx::agas::addressing_service::get_local_component_namespaces`
(C++ function), 1228 *(C++ function)*, 1216
`hpx::agas::addressing_service::component_id_type` `hpx::agas::addressing_service::get_local_locality`
(C++ type), 1215 *(C++ function)*, 1216
`hpx::agas::addressing_service::component_ns_` `hpx::agas::addressing_service::get_local_locality_namespaces`
(C++ member), 1229 *(C++ function)*, 1216
`hpx::agas::addressing_service::console_cache_` `hpx::agas::addressing_service::get_local_primary_namespace`
(C++ member), 1228 *(C++ function)*, 1216
`hpx::agas::addressing_service::console_cache_m` `hpx::agas::addressing_service::get_local_primary_namespace`
(C++ member), 1228 *(C++ function)*, 1216
`hpx::agas::addressing_service::decref` (*C++ function*), 1225 *(C++ function)*, 1216

hpx::agas::addressing_service::get_local_symbol
hpx::agas::addressing_service::iterate_types
 (C++ function), 1216
hpx::agas::addressing_service::get_localities hpx::agas::addressing_service::iterate_types_function_type
 (C++ function), 1217, 1218
hpx::agas::addressing_service::get_num_localities
 (C++ function), 1218
hpx::agas::addressing_service::get_num_localities
 (C++ function), 1218
hpx::agas::addressing_service::get_num_overall_threads
 (C++ function), 1218
hpx::agas::addressing_service::get_num_overall_threads
 (C++ function), 1218
hpx::agas::addressing_service::get_num_threads
hpx::agas::addressing_service::launch_bootstrap
 (C++ function), 1230
hpx::agas::addressing_service::locality_ns_
 (C++ member), 1229
hpx::agas::addressing_service::locality_ns_
 (C++ member), 1229
hpx::agas::addressing_service::mark_as_migrated
 (C++ function), 1228
hpx::agas::addressing_service::max_refcnt_requests_
 (C++ member), 1228
hpx::agas::addressing_service::migrated_objects_mtx_
 (C++ member), 1228
hpx::agas::addressing_service::migrated_objects_table_
 (C++ member), 1228
hpx::agas::addressing_service::migrated_objects_table_type
 (C++ type), 1215
hpx::agas::addressing_service::get_status
 (C++ function), 1216
hpx::agas::addressing_service::get_symbol_ns_l
hpx::agas::addressing_service::on_symbol_namespace_event
 (C++ function), 1216
hpx::agas::addressing_service::gva_cache_
 (C++ member), 1228
hpx::agas::addressing_service::gva_cache_mtx_ hpx::agas::addressing_service::primary_ns_
 (C++ member), 1228
hpx::agas::addressing_service::gva_cache_type hpx::agas::addressing_service::range_caching_
 (C++ type), 1215
hpx::agas::addressing_service::has_resolved_ld
hpx::agas::addressing_service::refcnt_requests_
 (C++ function), 1217
hpx::agas::addressing_service::HPX_NON_COPYABLE
hpx::agas::addressing_service::refcnt_requests_count_
 (C++ function), 1215
hpx::agas::addressing_service::inref
 (C++ function), 1225
hpx::agas::addressing_service::inref_async
 (C++ function), 1225
hpx::agas::addressing_service::initialize
 (C++ function), 1215
hpx::agas::addressing_service::is_bootstrap
 (C++ function), 1216
hpx::agas::addressing_service::is_connecting
 (C++ function), 1216
hpx::agas::addressing_service::is_console
 (C++ function), 1216
hpx::agas::addressing_service::is_local_address
 (C++ function), 1223
hpx::agas::addressing_service::is_local_lva_endpoints
 (C++ function), 1223
hpx::agas::addressing_service::iterate_ids
 (C++ function), 1225
hpx::agas::addressing_service::iterate_names
 (C++ type), 1215
hpx::agas::addressing_service::iterate_names
 (C++ type), 1215
hpx::agas::addressing_service::register_console
 (C++ function), 1216
hpx::agas::addressing_service::register_factory
 (C++ function), 1219
hpx::agas::addressing_service::register_locality
 (C++ function), 1217
hpx::agas::addressing_service::register_name
 (C++ function), 1225, 1226
hpx::agas::addressing_service::register_name_async
 (C++ function), 1226
hpx::agas::addressing_service::register_server_instances
 (C++ function), 1216
hpx::agas::addressing_service::remove_cache_entry
 (C++ function), 1227
hpx::agas::addressing_service::remove_resolved_locality
 (C++ function), 1217

```

hpx::agas::addressing_service::resolve_async hpx::agas::addressing_service::unbind_range_async
    (C++ function), 1224                               (C++ function), 1223
hpx::agas::addressing_service::resolve_cached hpx::agas::addressing_service::unbind_range_local
    (C++ function), 1224, 1225                         (C++ function), 1222
hpx::agas::addressing_service::resolve_full_asynch::agas::addressing_service::unmark_as_migrated
    (C++ function), 1224                               (C++ function), 1228
hpx::agas::addressing_service::resolve_full_ldata hpx::agas::addressing_service::unregister_locality
    (C++ function), 1224, 1225                         (C++ function), 1217
hpx::agas::addressing_service::resolve_full_pdata hpx::agas::addressing_service::unregister_name
    (C++ function), 1230                               (C++ function), 1226
hpx::agas::addressing_service::resolve_local   hpx::agas::addressing_service::unregister_name_async
    (C++ function), 1223, 1224                         (C++ function), 1226
hpx::agas::addressing_service::resolve_locally hpx::agas::addressing_service::update_cache_entry
    (C++ function), 1217                               (C++ function), 1227
hpx::agas::addressing_service::resolved_localities::agas::bind (C++ function), 1304, 1305
    (C++ member), 1229                               hpx::agas::bind_gid_local (C++ function), 1305
hpx::agas::addressing_service::resolved_localities::agas::bind_range_local (C++ function), 1305
    (C++ member), 1229                             hpx::agas::bootstrap_primary_namespace_gid
hpx::agas::addressing_service::resolved_localities_type (C++ function), 1231
    (C++ type), 1215                               hpx::agas::bootstrap_primary_namespace_id
hpx::agas::addressing_service::rts_lva_
    (C++ member), 1229                         hpx::agas::decref (C++ function), 1305
hpx::agas::addressing_service::runtime_type hpx::agas::destroy_component (C++ function),
    (C++ member), 1229                         1306
hpx::agas::addressing_service::send_refcnt_requests::agas::end_migration (C++ function), 1305
    (C++ function), 1230                         hpx::agas::find_symbols (C++ function), 1306
hpx::agas::addressing_service::send_refcnt_requests::agas::garbage_collect (C++ function), 1305
    (C++ function), 1230                         hpx::agas::garbage_collect_non_blocking
hpx::agas::addressing_service::send_refcnt_requests_no_blocking (C++ function), 1305
    (C++ function), 1230                         hpx::agas::get_all_locality_ids (C++ function),
hpx::agas::addressing_service::send_refcnt_requests_sync (C++ function), 1304
    (C++ function), 1230                         hpx::agas::get_colocation_id (C++ function),
hpx::agas::addressing_service::service_type hpx::agas::get_component_id (C++ function), 1306
    (C++ member), 1229                         1305
hpx::agas::addressing_service::set_local_locality hpx::agas::get_component_type_name (C++ function),
    (C++ function), 1216                         1304
hpx::agas::addressing_service::set_status   hpx::agas::get_console_locality (C++ function),
    (C++ function), 1216                         1305
hpx::agas::addressing_service::start_shutdown hpx::agas::get_locality (C++ function), 1304
    (C++ function), 1228                         hpx::agas::get_locality_id (C++ function), 1304
hpx::agas::addressing_service::state_   (C++ member), 1229                         hpx::agas::get_next_id (C++ function), 1305
hpx::agas::addressing_service::symbol_ns_
    (C++ member), 1229                         hpx::agas::get_num_localities (C++ function),
hpx::agas::addressing_service::synchronize_with_async::timer (C++ function), 1304
    (C++ function), 1216                         hpx::agas::get_num_overall_threads (C++ function)
hpx::agas::addressing_service::unbind_local (C++ function), 1221                         hpx::agas::get_num_threads (C++ function), 1304
hpx::agas::addressing_service::unbind_range (C++ function), 1223                         hpx::agas::get_primary_ns_lva (C++ function),
                                            1306

```


(C++ function), 1235
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::locality_ (C++ member), 1234
(C++ member), 1236
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::migrating_objects_ (C++ function), 1235
(C++ member), 1234
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::migration_table_type_ (C++ function), 1235
(C++ type), 1233
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::mutex_ (C++ function), 1235
(C++ member), 1234
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::mutex_type_ (C++ function), 1235
(C++ type), 1232
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::next_id_ (C++ function), 1235
(C++ member), 1234
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::primary_namespace (C++ member), 1236
(C++ function), 1233
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::refcnt_table_type_ (C++ member), 1236
(C++ type), 1232
hpx::agas::server::primary_namespace::counter hpx::agas::server::primary_namespace::refcnts_ (C++ member), 1233
(C++ function), 1234
hpx::agas::server::primary_namespace::decrement hpx::crefjats::server::primary_namespace::register_server_inst_ (C++ function), 1233
(C++ function), 1233
hpx::agas::server::primary_namespace::decrement hpx::swaps::server::primary_namespace::resolve_free_list_ (C++ function), 1234
(C++ function), 1234
hpx::agas::server::primary_namespace::end_migration hpx::agas::server::primary_namespace::resolve_gid_ (C++ function), 1233
(C++ function), 1233
hpx::agas::server::primary_namespace::finalize hpx::agas::server::primary_namespace::resolve_gid_locked_ (C++ function), 1233
(C++ function), 1234
hpx::agas::server::primary_namespace::free_component hpx::agas::server::primary_namespace::resolved_type_ (C++ function), 1234
(C++ type), 1232
hpx::agas::server::primary_namespace::free_entry hpx::agas::server::primary_namespace::set_local_locality_ (C++ struct), 1236
(C++ function), 1233
hpx::agas::server::primary_namespace::free_entry hpx::agas::server::primary_namespace::unbind_gid_ (C++ function), 1236
(C++ function), 1233
hpx::agas::server::primary_namespace::free_entry hpx::agas::server::primary_namespace::unregister_server_in_ (C++ member), 1237
(C++ function), 1233
hpx::agas::server::primary_namespace::free_entry hpx::agas::server::primary_namespace::wait_for_migration_l_ (C++ member), 1237
(C++ function), 1234
hpx::agas::server::primary_namespace::free_entry hpx::agas::server::primary_namespace::wait_for_migration_l_ (C++ member), 1237
(C++ function), 1232
hpx::agas::server::primary_namespace::free_entry hpx::all_of(C++ function), 1305
(C++ type), 1233
hpx::agas::server::primary_namespace::free_entry hpx::unbind_gid_local(C++ function), 1305
hpx::agas::server::primary_namespace::free_entry hpx::unbind_range_local(C++ function), 1305
(C++ type), 1233
hpx::agas::server::primary_namespace::gva_table hpx::data_type::unmark_as_migrated (C++ function), 1306
(C++ type), 1232
hpx::agas::server::primary_namespace::gva_table hpx::data_type::was_object_migrated (C++ function), 1303
(C++ type), 1232
hpx::agas::server::primary_namespace::gvas_ hpx::ags::unregister_name (C++ function), 1303
(C++ member), 1234
hpx::agas::server::primary_namespace::increment hpx::ags::update_cache_entry (C++ function), 1304
hpx::agas::server::primary_namespace::increment hpx::ags::was_object_migrated (C++ function), 1306
hpx::agas::server::primary_namespace::increment hpx::all_of(C++ function), 298, 299
hpx::agas::server::primary_namespace::increment hpx::create_tated_function (C++ function), 1166
(C++ function), 1234
hpx::agas::server::primary_namespace::instance hpx::any (C++ type), 970
hpx::agas::server::primary_namespace::instance hpx::any_cast (C++ function), 959

hpx::any_nonsr (*C++ type*), 958
hpx::any_of (*C++ function*), 296, 297
hpx::applier (*C++ type*), 1379, 1381
hpx::applier::applier (*C++ class*), 1379
hpx::applier::applier::~applier (*C++ function*),
 1380
hpx::applier::applier::applier (*C++ function*),
 1380
hpx::applier::applier::get_localities (*C++
 function*), 1381
hpx::applier::applier::get_locality_id (*C++
 function*), 1380
hpx::applier::applier::get_raw_localities
 (*C++ function*), 1380
hpx::applier::applier::get_raw_locality
 (*C++ function*), 1380
hpx::applier::applier::get_raw_remote_localities
 (*C++ function*), 1380
hpx::applier::applier::get_remote_localities hpx::binary_semaphore::acquire (*C++ function*),
 1112
 (*C++ function*), 1380
hpx::applier::applier::get_runtime_support_gid hpx::binary_semaphore::max (*C++ function*), 1113
 (*C++ function*), 1381
hpx::applier::applier::get_runtime_support_raw_gid
 (*C++ function*), 1381
hpx::applier::applier::get_thread_manager
 (*C++ function*), 1380
hpx::applier::applier::HPX_NON_COPYABLE
 (*C++ function*), 1380
hpx::applier::applier::init (*C++ function*), 1380
hpx::applier::applier::initialize (*C++ func-
 tion*), 1380
hpx::applier::applier::runtime_support_id_
 (*C++ member*), 1381
hpx::applier::applier::thread_manager_ (*C++
 member*), 1381
hpx::applier::get_applier (*C++ function*), 1381
hpx::applier::get_applier_ptr (*C++ function*),
 1381
hpx::assertion (*C++ type*), 885, 887
hpx::assertion::assertion_handler (*C++ type*),
 888
hpx::assertion::instead (*C++ type*), 887
hpx::assertion::set_assertion_handler (*C++
 function*), 888
hpx::async (*C++ function*), 888
hpx::bad_any_cast (*C++ struct*), 959
hpx::bad_any_cast::bad_any_cast (*C++ function*),
 960
hpx::bad_any_cast::from (*C++ member*), 960
hpx::bad_any_cast::to (*C++ member*), 960
hpx::bad_any_cast::what (*C++ function*), 960
hpx::barrier (*C++ class*), 1109
hpx::barrier::arrival_token (*C++ type*), 1110
hpx::barrier::arrive (*C++ function*), 1110
hpx::barrier::arrive_and_drop (*C++ function*),
 1110
hpx::barrier::arrive_and_wait (*C++ function*),
 1110
hpx::barrier::arrived_ (*C++ member*), 1111
hpx::barrier::barrier (*C++ function*), 1110
hpx::barrier::completion_ (*C++ member*), 1111
hpx::barrier::cond_ (*C++ member*), 1111
hpx::barrier::expected_ (*C++ member*), 1111
hpx::barrier::mtx_ (*C++ member*), 1111
hpx::barrier::mutex_type (*C++ type*), 1111
hpx::barrier::phase_ (*C++ member*), 1111
hpx::barrier::wait (*C++ function*), 1110
hpx::binary_semaphore (*C++ class*), 1111
hpx::binary_semaphore::~binary_semaphore
 (*C++ function*), 1112
hpx::binary_semaphore::binary_semaphore
 (*C++ function*), 1112
hpx::binary_semaphore::operator= (*C++ func-
 tion*), 1112
hpx::binary_semaphore::release (*C++ function*),
 1112
hpx::binary_semaphore::try_acquire (*C++ func-
 tion*), 1112
hpx::binary_semaphore::try_acquire_for (*C++
 function*), 1112
hpx::binary_semaphore::try_acquire_until
 (*C++ function*), 1112
hpx::bind (*C++ function*), 1029
hpx::bind_front (*C++ function*), 1032
hpx::call_once (*C++ function*), 1143
hpx::chrono (*C++ type*), 1196, 1197
hpx::chrono::high_resolution_clock (*C++
 struct*), 1196
hpx::chrono::high_resolution_clock::now
 (*C++ function*), 1197
hpx::chrono::high_resolution_timer (*C++
 class*), 1197
hpx::chrono::high_resolution_timer::elapsed
 (*C++ function*), 1197
hpx::chrono::high_resolution_timer::elapsed_max
 (*C++ function*), 1198
hpx::chrono::high_resolution_timer::elapsed_microseconds
 (*C++ function*), 1197
hpx::chrono::high_resolution_timer::elapsed_min
 (*C++ function*), 1198
hpx::chrono::high_resolution_timer::elapsed_nanoseconds
 (*C++ function*), 1197
hpx::chrono::high_resolution_timer::high_resolution_timer
 (*C++ function*), 1197
hpx::chrono::high_resolution_timer::init

(C++ enum), 1197
hpx::chrono::high_resolution_timer::init::no_imix (C++ enumerator), 1197
hpx::chrono::high_resolution_timer::now (C++ function), 1198
hpx::chrono::high_resolution_timer::restart (C++ function), 1197
hpx::chrono::high_resolution_timer::start_time (C++ member), 1198
hpx::chrono::high_resolution_timer::take_time (C++ function), 1198
hpx::collectives (C++ type), 1264, 1265, 1267, 1268, 1272, 1277–1279, 1283, 1286, 1289, 1293
hpx::collectives::all_gather (C++ function), 1264, 1265
hpx::collectives::all_reduce (C++ function), 1266
hpx::collectives::all_to_all (C++ function), 1267, 1268
hpx::collectives::broadcast_from (C++ function), 1273, 1274
hpx::collectives::broadcast_to (C++ function), 1272, 1273
hpx::collectives::create_channel_communicator (C++ function), 1277
hpx::collectives::create_communicator (C++ function), 1279
hpx::collectives::exclusive_scan (C++ function), 1279, 1280
hpx::collectives::gather_here (C++ function), 1283, 1284
hpx::collectives::gather_there (C++ function), 1284, 1285
hpx::collectives::generation_arg (C++ struct), 1268
hpx::collectives::generation_arg::generation (C++ member), 1269
hpx::collectives::generation_arg::generation_arg (C++ function), 1269
hpx::collectives::generation_arg::operator std::size_t (C++ function), 1269
hpx::collectives::generation_arg::operator= (C++ function), 1269
hpx::collectives::get (C++ function), 1278
hpx::collectives::inclusive_scan (C++ function), 1286, 1287
hpx::collectives::num_sites_arg (C++ struct), 1269
hpx::collectives::num_sites_arg::num_sites_ (C++ member), 1269
hpx::collectives::num_sites_arg::num_sites_arg (C++ function), 1269
hpx::collectives::num_sites_arg::operator
std::size_t (C++ function), 1269
hpx::collectives::operator= (C++ function), 1269
hpx::collectives::reduce_here (C++ function), 1289, 1290
hpx::collectives::reduce_there (C++ function), 1290, 1291
hpx::collectives::root_site_arg (C++ struct), 1269
hpx::collectives::root_site_arg::operator std::size_t (C++ function), 1269
hpx::collectives::root_site_arg::operator= (C++ function), 1269
hpx::collectives::root_site_arg::root_site_ (C++ member), 1269
hpx::collectives::root_site_arg::root_site_arg (C++ function), 1269
hpx::collectives::scatter_from (C++ function), 1293, 1294
hpx::collectives::scatter_to (C++ function), 1294, 1295
hpx::collectives::set (C++ function), 1277
hpx::collectives::tag_arg (C++ struct), 1269
hpx::collectives::tag_arg::operator std::size_t (C++ function), 1270
hpx::collectives::tag_arg::operator= (C++ function), 1270
hpx::collectives::tag_arg::tag_ (C++ member), 1270
hpx::collectives::tag_arg::tag_arg (C++ function), 1270
hpx::collectives::that_site_arg (C++ struct), 1270
hpx::collectives::that_site_arg::operator std::size_t (C++ function), 1270
hpx::collectives::that_site_arg::operator= (C++ function), 1270
hpx::collectives::that_site_arg::that_site_ (C++ member), 1270
hpx::collectives::that_site_arg::that_site_arg (C++ function), 1270
hpx::collectives::this_site_arg (C++ struct), 1270
hpx::collectives::this_site_arg::operator std::size_t (C++ function), 1270
hpx::collectives::this_site_arg::operator= (C++ function), 1270
hpx::collectives::this_site_arg::this_site_ (C++ member), 1270
hpx::collectives::this_site_arg::this_site_arg (C++ function), 1270

1391–1393, 1398
hpx::components::abstract_component_base 1311
 (C++ class), 1311
hpx::components::abstract_managed_component_base 1306
 (C++ class), 1311
hpx::components::binpacked (C++ member), 1317
hpx::components::binpacking_distribution_policy 1077
 (C++ struct), 1317
hpx::components::binpacking_distribution_policy::binpacking_distribution_policy 1307
 (C++ function), 1317
hpx::components::binpacking_distribution_policy::bulk (Create function), 1077
 (C++ function), 1318
hpx::components::binpacking_distribution_policy::create (C++ type), 1309
 (C++ function), 1317
hpx::components::binpacking_distribution_policy::get_counter 1309
 (C++ function), 1318
hpx::components::binpacking_distribution_policy::get_mutable_localities 1309
 (C++ function), 1318
hpx::components::binpacking_distribution_policy::operator<< (C++ enumerator), 1309
 (C++ function), 1317
hpx::components::client_base (C++ class), 1300
hpx::components::colocated (C++ member), 1318
hpx::components::colocating_distribution_policy 1309
 (C++ struct), 1318
hpx::components::colocating_distribution_policy::apply (C++ enumerator), 1309
 (C++ function), 1319, 1320
hpx::components::colocating_distribution_policy::apply (C++ enumerator), 1309
 (C++ function), 1320
hpx::components::colocating_distribution_policy::async (C++ enumerator), 1309
 (C++ function), 1319
hpx::components::colocating_distribution_policy::async (C++ enumerator), 1309
 (C++ function), 1319
hpx::components::colocating_distribution_policy::async (Create enumerator), 1309
 (C++ struct), 1320
hpx::components::colocating_distribution_policy::async (Create type), 1309
 (C++ type), 1320
hpx::components::colocating_distribution_policy::bulk (Create enumerator), 1309
 (C++ function), 1319
hpx::components::colocating_distribution_policy::colocating_distribution_policy 1309
 (C++ function), 1319
hpx::components::colocating_distribution_policy::create (C++ enumerator), 1309
 (C++ function), 1319
hpx::components::colocating_distribution_policy::get_next_targetator 1309
 (C++ function), 1320
hpx::components::colocating_distribution_policy::get_mutable_localities 1309
 (C++ function), 1320
hpx::components::colocating_distribution_policy::operator<< (C++ enumerator), 1309
 (C++ function), 1319
hpx::components::commandline_options_provider 1367
 (C++ type), 1307
hpx::components::commandline_options_provider::add_commandline_options 1367
 (C++ function), 1307
hpx::components::component (C++ class), 1311
hpx::components::component_base (C++ class), 1309
 (C++ struct), 1309
hpx::components::component_commandline (C++ struct), 1306
hpx::components::component_commandline::add_commandline_options 1307
 (C++ function), 1307
hpx::components::component_commandline_base 1077
 (C++ struct), 1077
hpx::components::component_commandline_base::~component_commandline_base 1309
hpx::components::component_deleter_type 1309
hpx::components::component_enum_type (C++ type), 1309
hpx::components::component_enum_type (C++ function), 1309
hpx::components::component_agas_components 1309
hpx::components::component_agas_locals 1309
hpx::components::component_agas_locals::component_agas_local 1309
hpx::components::component_agas_locals::operator<< (C++ enumerator), 1309
hpx::components::component_agas_locals::operator<< (C++ type), 1309
hpx::components::component_agas_locals::operator<< (Create enumerator), 1309
hpx::components::component_agas_locals::operator<< (Create type), 1309
hpx::components::component_agas_locals::operator<< (Plain function), 1309
hpx::components::component_agas_locals::operator<< (Upper bound), 1309
hpx::components::component_agas_locals::operator<< (Factory), 1309
hpx::components::component_agas_locals::operator<< (Type), 1309
hpx::components::component_agas_locals::operator<< (Function), 1309
hpx::components::component_agas_locals::operator<< (Struct), 1309
hpx::components::component_agas_locals::operator<< (Registry), 1309
hpx::components::component_agas_locals::operator<< (Info), 1309
hpx::components::component_agas_locals::register_component_type 1309

(*C++ function*), 1315
hpx::components::detail_adl_barrier::PhonyNameToComponentName (C++ function), 1316
hpx::components::enabled (*C++ function*), 1310
hpx::components::enumerate_instance_counts (*C++ function*), 1310
hpx::components::factory_state_enum (*C++ enum*), 1309
hpx::components::factory_state_enum::factory_check (*C++ enumerator*), 1310
hpx::components::factory_state_enum::factory_disabled (*C++ enumerator*), 1310
hpx::components::factory_state_enum::factory_enabled (*member*), 1392
hpx::components::fixed_component (*C++ class*), 1311, 1312
hpx::components::fixed_component_base (*C++ class*), 1311, 1312
hpx::components::get_base_type (*C++ function*), 1310
hpx::components::get_component_base_name (*C++ function*), 1310
hpx::components::get_component_name (*C++ function*), 1310
hpx::components::get_component_type (*C++ function*), 1310
hpx::components::get_component_type_name (*C++ function*), 1310
hpx::components::get_derived_type (*C++ function*), 1310
hpx::components::instance_count (*C++ function*), 1310
hpx::components::instead (*C++ type*), 1311
hpx::components::intrusive_ptr_add_ref (*C++ function*), 1314
hpx::components::intrusive_ptr_release (*C++ function*), 1314
hpx::components::managed_component (*C++ class*), 1311, 1314
hpx::components::managed_component_base (*C++ class*), 1311, 1314
hpx::components::migrate (*C++ function*), 1389, 1390
hpx::components::runtime_support (*C++ class*), 1391
hpx::components::runtime_support::base_type (*C++ type*), 1392
hpx::components::runtime_support::bulk_create_libraries (C++ function), 1391
hpx::components::runtime_support::bulk_create_libraries::server::runtime_support::add_shutdown_function (C++ function), 1391
hpx::components::runtime_support::call_startupfunctions (C++ function), 1392
hpx::components::runtime_support::call_startupfunctions::server::runtime_support::add_startup_function (C++ function), 1392
hpx::components::runtime_support::call_startupfunctions::server::runtime_support::bulk_create_com-

(*C++ function*), 1392
hpx::components::runtime_support::create_component (C++ function), 1391
hpx::components::runtime_support::create_component_async (C++ function), 1391
hpx::components::runtime_support::get_config (C++ function), 1392
hpx::components::runtime_support::get_id (C++ function), 1392
hpx::components::runtime_support::get_raw_gid (C++ function), 1392
hpx::components::runtime_support::gid_ (*C++ member*) (C++ function), 1392
hpx::components::runtime_support::load_components (C++ function), 1392
hpx::components::runtime_support::load_components_async (C++ function), 1391
hpx::components::runtime_support::runtime_support (C++ function), 1391
hpx::components::runtime_support::shutdown (C++ function), 1392
hpx::components::runtime_support::shutdown_all (C++ function), 1392
hpx::components::runtime_support::shutdown_async (C++ function), 1392
hpx::components::runtime_support::terminate (C++ function), 1392
hpx::components::runtime_support::terminate_all (C++ function), 1392
hpx::components::runtime_support::terminate_async (C++ function), 1392
hpx::components::server (*C++ type*), 1367, 1392, 1393
hpx::components::server::copy_component (C++ function), 1393
hpx::components::server::copy_component_action (*C++ struct*), 1393
hpx::components::server::copy_component_action_here (*C++ struct*), 1393
hpx::components::server::copy_component_here (C++ function), 1393
hpx::components::server::runtime_support (*C++ class*), 1393
hpx::components::server::runtime_support::~runtime_support (C++ function), 1393
hpx::components::server::runtime_support::add_pre_shutdown_function (C++ function), 1395
hpx::components::server::runtime_support::add_shutdown_function (C++ function), 1395
hpx::components::server::runtime_support::add_startup_function (C++ function), 1395
hpx::components::server::runtime_support::bulk_create_com-

```

(C++ function), 1394
hpx::components::server::runtime_support::callhpx::shutdown_componentshpx::start_componentsserver::runtime_support::modules_map_type
(C++ function), 1394
hpx::components::server::runtime_support::callhpx::start_componentshpx::shutdown_componentsserver::runtime_support::mtx_
(C++ function), 1394
hpx::components::server::runtime_support::copyhpx::create_componenthpx::create_componentserver::runtime_support::notify_waiting_m
(C++ function), 1394
hpx::components::server::runtime_support::createhpx::componentshpx::componentsserver::runtime_support::p_mtx_
(C++ function), 1393, 1394
hpx::components::server::runtime_support::createhpx::perfmonceshpx::perfmoncesserver::runtime_support::plugin_factory
(C++ function), 1394
hpx::components::server::runtime_support::deletehpx::function_listhpx::function_listserver::runtime_support::plugin_factory::
(C++ function), 1393
hpx::components::server::runtime_support::dijkstrahpx::termination_selectionhpx::termination_selectionserver::runtime_support::plugin_factory::
(C++ function), 1397
hpx::components::server::runtime_support::finalizehpx::componentshpx::componentsserver::runtime_support::plugin_factory::
(C++ function), 1395
hpx::components::server::runtime_support::garbagehpx::componentshpx::componentsserver::runtime_support::plugin_factory::
(C++ function), 1394
hpx::components::server::runtime_support::gethpx::component_typeshpx::component_typesserver::runtime_support::plugin_factory_t
(C++ function), 1395
hpx::components::server::runtime_support::gethpx::confighpx::configcomponents::server::runtime_support::plugin_map_mutex
(C++ function), 1394
hpx::components::server::runtime_support::globalhpx::componentshpx::componentsserver::runtime_support::plugin_map_type
(C++ member), 1398
(C++ type), 1397
hpx::components::server::runtime_support::ishpx::targethpx::targetcomponents::server::runtime_support::plugins_
(C++ function), 1396
hpx::components::server::runtime_support::loadhpx::component_opthpx::component_optserver::runtime_support::pre_shutdown_func
(C++ function), 1396
hpx::components::server::runtime_support::loadhpx::component_opthpx::component_optserver::runtime_support::pre_startup_func
(C++ function), 1396
hpx::components::server::runtime_support::loadhpx::componentshpx::componentsserver::runtime_support::remove_from_conn
(C++ function), 1396
hpx::components::server::runtime_support::loadhpx::component_opthpx::component_optserver::runtime_support::remove_here_from_
(C++ function), 1396
hpx::components::server::runtime_support::loadhpx::component_stathpx::component_statserver::runtime_support::remove_here_from_
(C++ function), 1396
hpx::components::server::runtime_support::loadhpx::componentshpx::componentsserver::runtime_support::runtime_support
(C++ function), 1394, 1396
(C++ function), 1393
hpx::components::server::runtime_support::loadhpx::plugin_componentshpx::plugin_componentsserver::runtime_support::set_component_ty
(C++ function), 1396
hpx::components::server::runtime_support::loadhpx::plugin_componentshpx::plugin_componentsserver::runtime_support::shutdown
(C++ function), 1397
(C++ function), 1394
hpx::components::server::runtime_support::loadhpx::plugin_componentshpx::plugin_componentsserver::runtime_support::shutdown_all
(C++ function), 1396
(C++ function), 1394
hpx::components::server::runtime_support::loadhpx::start_componentshpx::start_componentsserver::runtime_support::shutdown_all_inv
(C++ function), 1396
(C++ member), 1397
hpx::components::server::runtime_support::loadhpx::start_componentshpx::start_componentsserver::runtime_support::shutdown_function
(C++ function), 1396
(C++ member), 1398
hpx::components::server::runtime_support::mainhpx::threadshpx::threadscomponents::server::runtime_support::startup_function
(C++ member), 1397
hpx::components::server::runtime_support::migratehpx::componentshpx::componentstohpx::serverhpx::server::runtime_support::static_modules_
(C++ function), 1394
(C++ member), 1398
hpx::components::server::runtime_support::modulehpx::componentshpx::componentsserver::runtime_support::static_modules_t

```

(C++ type), 1397
hpx::components::server::runtime_support::stop (C++ function), 1395
hpx::components::server::runtime_support::stop (C++ member), 1397
hpx::components::server::runtime_support::stop_called (C++ function), 1400
hpx::components::server::runtime_support::stop_conditional (C++ function), 1399
hpx::components::server::runtime_support::stop_done_ (C++ function), 1401
hpx::components::server::runtime_support::stop (C++ member), 1397
hpx::components::server::runtime_support::stopped (C++ function), 1401
hpx::components::server::runtime_support::terminated (C++ function), 1395
hpx::components::server::runtime_support::terminated_a (C++ function), 1394
hpx::components::server::runtime_support::terminated_all (C++ function), 1394
hpx::components::server::runtime_support::terminated_a (C++ function), 1394
hpx::components::server::runtime_support::terminated_all (C++ function), 1394
hpx::components::server::runtime_support::terminated_(C++ function), 1401
hpx::components::server::runtime_support::terminated_(C++ member), 1397
hpx::components::server::runtime_support::tidy (C++ function), 1400
hpx::components::server::runtime_support::type_holder (C++ function), 1400
hpx::components::server::runtime_support::type (C++ type), 1393
hpx::components::server::runtime_support::wait (C++ function), 1395
hpx::components::server::runtime_support::wait (C++ member), 1397
hpx::components::server::runtime_support::wait_conditional (C++ function), 1400
hpx::components::server::runtime_support::was_stopped (C++ function), 1401
hpx::components::set_component_type (C++ function), 1310
hpx::components::stubs (C++ type), 1367, 1398
hpx::components::stubs::runtime_support (C++ struct), 1398
hpx::components::stubs::runtime_support::bulk_create_component (C++ function), 1399
hpx::components::stubs::runtime_support::bulk_create_component::stubs (C++ type), 1399
hpx::components::stubs::runtime_support::bulk_create_component::stubs::runtime_support::terminate (C++ function), 1401
hpx::components::stubs::runtime_support::bulk_create_component::stubs::runtime_support::terminate_all (C++ function), 1399
hpx::components::stubs::runtime_support::bulk_create_component::stubs::runtime_support::terminate_async (C++ function), 1399
hpx::components::stubs::runtime_support::bulk_create_component::targeted_asynch (C++ function), 1321
hpx::components::stubs::runtime_support::call_startup (functions), 1321
hpx::components::stubs::runtime_support::call_startup (function), 1400
hpx::components::stubs::runtime_support::call_startup (function), 1322
hpx::components::stubs::runtime_support::copy_create (implementation), 1322
hpx::components::stubs::runtime_support::copy_create (C++ function), 1400
hpx::components::stubs::runtime_support::copy_create (implementation), 1321
hpx::components::stubs::runtime_support::copy_create (C++ function), 1400
hpx::components::stubs::runtime_support::create_component (C++ function), 1321
hpx::components::stubs::runtime_support::create_component (C++ function), 1399
hpx::components::stubs::runtime_support::create_component (C++ function), 1399
hpx::components::stubs::runtime_support::create_component (C++ function), 1399
hpx::components::stubs::runtime_support::create_component (C++ function), 1399
hpx::components::stubs::runtime_support::create_performance (C++ function), 1401
hpx::components::stubs::runtime_support::create_result (C++ function), 1399
hpx::components::stubs::runtime_support::garbage_collect (C++ function), 1401
hpx::components::stubs::runtime_support::get_config (C++ function), 1401
hpx::components::stubs::runtime_support::get_config_async (C++ function), 1401
hpx::components::stubs::runtime_support::load_components (C++ function), 1401
hpx::components::stubs::runtime_support::load_components_all (C++ function), 1401
hpx::components::stubs::runtime_support::migrate_component (C++ function), 1400
hpx::components::stubs::runtime_support::remove_from_connection (C++ function), 1401
hpx::components::stubs::runtime_support::shutdown (C++ function), 1401
hpx::components::stubs::runtime_support::shutdown_all (C++ function), 1400
hpx::components::stubs::runtime_support::shutdown_async (C++ function), 1400
hpx::components::target_distribution_policy (C++ function), 1321
hpx::components::target_distribution_policy::apply (C++ function), 1400
hpx::components::target_distribution_policy::apply_cb (C++ function), 1322
hpx::components::target_distribution_policy::async (C++ function), 1321
hpx::components::target_distribution_policy::async_cb (C++ function), 1400
hpx::components::target_distribution_policy::async_result (C++ function), 1399

(C++ struct), 1322
`hpx::components::target_distribution_policy::async_comPUTE::type::block_executor::operator=`
(C++ type), 1322
`hpx::components::target_distribution_policy::bulk_comPUTE::host::block_executor::priority_`
(C++ function), 1321
`hpx::components::target_distribution_policy::computeComPUTE::host::block_executor::scheduleHint_`
(C++ function), 1321
`hpx::components::target_distribution_policy::dEx_nExComputeHost::block_executor::stacksize_`
(C++ function), 1322
`hpx::components::target_distribution_policy::dEx_nUmPdLdHost::block_executor::tag_invoke`
(C++ function), 1322
`hpx::components::target_distribution_policy::operatorComPUTE::host::block_executor::targets`
(C++ function), 1321
`hpx::components::target_distribution_policy::targetDistributionPolicyExecutor::targets_`
(C++ function), 1321
`hpx::components::types_are_compatible` (C++ function), 1310
`hpx::components::unwrapping_result_policy` (C++ struct), 1322
`hpx::components::unwrapping_result_policy::apply` (C++ function), 1323
`hpx::components::unwrapping_result_policy::apply` (C++ function), 1323
`hpx::components::unwrapping_result_policy::async` (C++ function), 1323
`hpx::components::unwrapping_result_policy::async_cb` (C++ function), 1323
`hpx::components::unwrapping_result_policy::async` (C++ struct), 1323
`hpx::components::unwrapping_result_policy::async` (C++ type), 1324
`hpx::components::unwrapping_result_policy::get` (C++ function), 1323
`hpx::components::unwrapping_result_policy::unwappiComPmEltyPolicy::clear` (C++ function), 1323
`hpx::compute` (C++ type), 935, 939
`hpx::compute::host` (C++ type), 939
`hpx::compute::host::block_executor` (C++ struct), 939
`hpx::compute::host::block_executor::block_executor` (C++ function), 940
`hpx::compute::host::block_executor::bulk_async_execute` (C++ function), 940
`hpx::compute::host::block_executor::bulk_sync_execute` (C++ function), 940
`hpx::compute::host::block_executor::current_` (C++ member), 941
`hpx::compute::host::block_executor::executor_parameters` (C++ type), 940
`hpx::compute::host::block_executor::executors` (C++ member), 941
`hpx::compute::host::block_executor::get_next_executor` (C++ function), 940
`hpx::compute::host::block_executor::init_executor` (C++ function), 940
`hpx::compute::operator=` (C++ function), 941
`hpx::compute::swap` (C++ function), 936
`hpx::compute::vector` (C++ class), 936
`hpx::compute::vector::~vector` (C++ function), 937
`hpx::compute::vector::access_target` (C++ type), 936
`hpx::compute::vector::alloc` (C++ member), 938
`hpx::compute::vector::alloc_traits` (C++ type), 938
`hpx::compute::vector::allocator_type` (C++ type), 938
`hpx::compute::vector::begin` (C++ function), 938
`hpx::compute::vector::capacity` (C++ function), 937
`hpx::compute::vector::capacity_` (C++ member), 938
`hpx::compute::vector::cbegin` (C++ function), 938
`hpx::compute::vector::cend` (C++ function), 938
`hpx::compute::vector::const_iterator` (C++ type), 936
`hpx::compute::vector::const_pointer` (C++ type), 936
`hpx::compute::vector::const_reference` (C++ type), 936
`hpx::compute::vector::const_reverse_iterator` (C++ type), 936
`hpx::compute::vector::empty` (C++ function), 937
`hpx::compute::vector::end` (C++ function), 938
`hpx::compute::vector::get_allocator` (C++ function), 938
`hpx::compute::vector::iterator` (C++ type), 936
`hpx::compute::vector::operator=` (C++ function), 936

937
hpx::compute::vector::operator[] (C++ function), 937
hpx::compute::vector::pointer (C++ type), 936
hpx::compute::vector::reference (C++ type), 936
hpx::compute::vector::resize (C++ function), 937, 938
hpx::compute::vector::reverse_iterator (C++ type), 936
hpx::compute::vector::size (C++ function), 937
hpx::compute::vector::size_ (C++ member), 938
hpx::compute::vector::size_type (C++ type), 936
hpx::compute::vector::swap (C++ function), 938
hpx::compute::vector::value_type (C++ type), 936
hpx::compute::vector::vector (C++ function), 937
hpx::condition_variable (C++ class), 1113
hpx::condition_variable::~condition_variable (C++ function), 1114
hpx::condition_variable::condition_variable (C++ function), 1114
hpx::condition_variable::data_ (C++ member), 1118
hpx::condition_variable::data_type (C++ type), 1118
hpx::condition_variable::mutex_type (C++ type), 1118
hpx::condition_variable::notify_all (C++ function), 1115
hpx::condition_variable::notify_one (C++ function), 1114
hpx::condition_variable::wait (C++ function), 1115
hpx::condition_variable::wait_for (C++ function), 1117
hpx::condition_variable::wait_until (C++ function), 1116
hpx::condition_variable_any (C++ class), 1118
hpx::condition_variable_any::~condition_variable_any (C++ function), 1119
hpx::condition_variable_any::condition_variable (C++ function), 1119
hpx::condition_variable_any::data_ (C++ member), 1126
hpx::condition_variable_any::data_type (C++ type), 1126
hpx::condition_variable_any::mutex_type (C++ type), 1126
hpx::condition_variable_any::notify_all (C++ function), 1119
hpx::condition_variable_any::notify_one (C++ function), 1119
hpx::condition_variable_any::wait (C++ function), 1120, 1124
hpx::condition_variable_any::wait_for (C++ function), 1122, 1123, 1125
hpx::condition_variable_any::wait_until (C++ function), 1121, 1122, 1124
hpx::copy (C++ function), 300
hpx::copy_if (C++ function), 302, 303
hpx::copy_n (C++ function), 301, 302
hpx::count (C++ function), 304, 305
hpx::count_if (C++ function), 305, 307
hpx::counting_semaphore (C++ class), 1127
hpx::counting_semaphore::~counting_semaphore (C++ function), 1128
hpx::counting_semaphore::acquire (C++ function), 1128
hpx::counting_semaphore::counting_semaphore (C++ function), 1128
hpx::counting_semaphore::max (C++ function), 1129
hpx::counting_semaphore::operator= (C++ function), 1128
hpx::counting_semaphore::release (C++ function), 1128
hpx::counting_semaphore::try_acquire (C++ function), 1128
hpx::counting_semaphore::try_acquire_for (C++ function), 1128
hpx::counting_semaphore::try_acquire_until (C++ function), 1128
hpx::counting_semaphore_var (C++ class), 1129
hpx::counting_semaphore_var::acquire (C++ function), 1130
hpx::counting_semaphore_var::counting_semaphore_var (C++ function), 1130
hpx::counting_semaphore_var::max (C++ function), 1131
hpx::counting_semaphore_var::mutex_type (C++ type), 1131
hpx::counting_semaphore_var::operator= (C++ function), 1130
hpx::counting_semaphore_var::release (C++ function), 1130
hpx::counting_semaphore_var::signal (C++ function), 1130
hpx::counting_semaphore_var::signal_all (C++ function), 1130
hpx::counting_semaphore_var::try_acquire (C++ function), 1130
hpx::counting_semaphore_var::try_acquire_for (C++ function), 1131
hpx::counting_semaphore_var::try_acquire_until (C++ function), 1131
hpx::counting_semaphore_var::try_wait (C++ function), 1130
hpx::counting_semaphore_var::wait (C++ func-

tion), 1130
`hpx::cuda (C++ type)`, 910
`hpx::cuda::experimental (C++ type)`, 910
`hpx::cuda::experimental::cuda_executor (C++ struct)`, 910
`hpx::cuda::experimental::cuda_executor::~cuda_executor (C++ function)`, 910
`hpx::cuda::experimental::cuda_executor::apply (C++ function)`, 911
`hpx::cuda::experimental::cuda_executor::async (C++ function)`, 911
`hpx::cuda::experimental::cuda_executor::cuda_executor (C++ function)`, 910
`hpx::cuda::experimental::cuda_executor::tag_invoke (C++ function)`, 910
`hpx::cuda::experimental::cuda_executor_base (C++ struct)`, 911
`hpx::cuda::experimental::cuda_executor_base::cuda_executor_base (C++ function)`, 911
`hpx::cuda::experimental::cuda_executor_base::device_ (C++ member)`, 911
`hpx::cuda::experimental::cuda_executor_base::event_mode (C++ type)`, 911
`hpx::cuda::experimental::cuda_executor_base::future_type (C++ type)`, 911
`hpx::cuda::experimental::cuda_executor_base::get_future (C++ function)`, 911
`hpx::cuda::experimental::cuda_executor_base::promise (C++ member)`, 911
`hpx::cuda::experimental::cuda_executor_base::swap (C++ member)`, 911
`hpx::custom_exception_info_handler_type (C++ type)`, 980
`hpx::cv_status (C++ enum)`, 1113
`hpx::cv_status::error (C++ enumerator)`, 1113
`hpx::cv_status::no_timeout (C++ enumerator)`, 1113
`hpx::cv_status::timeout (C++ enumerator)`, 1113
`hpx::dataflow (C++ function)`, 889
`hpx::destroy (C++ function)`, 308
`hpx::destroy_n (C++ function)`, 308, 309
`hpx::diagnostic_information (C++ function)`, 1082
`hpx::disconnect (C++ function)`, 1326, 1327
`hpx::distributed (C++ type)`, 1034, 1041, 1243, 1248, 1271, 1287
`hpx::distributed::barrier (C++ class)`, 1271
`hpx::distributed::barrier::barrier (C++ function)`, 1271
`hpx::distributed::barrier::synchronize (C++ function)`, 1272
`hpx::distributed::barrier::wait (C++ function)`, 1272
`hpx::distributed::function (C++ type)`, 1034
`hpx::distributed::latch (C++ class)`, 1287
`hpx::distributed::latch::arrive_and_wait (C++ function)`, 1288
`hpx::distributed::latch::base_type (C++ type)`, 1289
`hpx::distributed::latch::count_down (C++ function)`, 1288
`hpx::distributed::latch::count_down_and_wait (C++ function)`, 1288
`hpx::distributed::latch::is_ready (C++ function)`, 1288
`hpx::distributed::latch::latch (C++ function)`, 1288
`hpx::distributed::latch::try_wait (C++ function)`, 1288
`hpx::distributed::latch::wait (C++ function)`, 1288
`hpx::distributed::move_only_function (C++ type)`, 1288
`hpx::distributed::PhonyNameDueToError::~promise`
`hpx::distributed::PhonyNameDueToError::base_type`
`hpx::distributed::PhonyNameDueToError::operator= (C++ function)`, 1249
`hpx::distributed::PhonyNameDueToError::promise`
`hpx::distributed::PhonyNameDueToError::set_value (C++ function)`, 1249
`hpx::distributed::PhonyNameDueToError::swap (C++ function)`, 1249
`hpx::distributed::promise (C++ class)`, 1243, 1248
`hpx::distributed::promise<void, hpx::util::unused_type> (C++ class)`, 1247
`hpx::distributed::promise<void, hpx::util::unused_type>::~promise (C++ function)`, 1248
`hpx::distributed::promise<void, hpx::util::unused_type>::base_type (C++ type)`, 1248
`hpx::distributed::promise<void, hpx::util::unused_type>::operator= (C++ function)`, 1248
`hpx::distributed::promise<void, hpx::util::unused_type>::promise (C++ function)`, 1247
`hpx::distributed::promise<void, hpx::util::unused_type>::set_value (C++ function)`, 1248
`hpx::distributed::promise<void, hpx::util::unused_type>::swap (C++ function)`, 1248
`hpx::distributed::swap (C++ function)`, 1248

hpx::ends_with (C++ function), 310
hpx::enumerate_os_threads (C++ function), 1098
hpx::equal (C++ function), 312–317
hpx::error (C++ enum), 973
hpx::error::assertion_failure (C++ enumerator), 974
hpx::error::bad_action_code (C++ enumerator), 973
hpx::error::bad_component_type (C++ enumerator), 973
hpx::error::bad_function_call (C++ enumerator), 976
hpx::error::bad_parameter (C++ enumerator), 974
hpx::error::bad_plugin_type (C++ enumerator), 976
hpx::error::bad_request (C++ enumerator), 974
hpx::error::bad_response_type (C++ enumerator), 974
hpx::error::broken.promise (C++ enumerator), 975
hpx::error::broken_task (C++ enumerator), 975
hpx::error::commandline_option_error (C++ enumerator), 975
hpx::error::deadlock (C++ enumerator), 974
hpx::error::duplicate_component_address (C++ enumerator), 974
hpx::error::duplicate_component_id (C++ enumerator), 976
hpx::error::duplicate_console (C++ enumerator), 974
hpx::error::dynamic_link_failure (C++ enumerator), 975
hpx::error::filesystem_error (C++ enumerator), 976
hpx::error::future_already_retrieved (C++ enumerator), 975
hpx::error::future_can_not_be_cancelled (C++ enumerator), 975
hpx::error::future_cancelled (C++ enumerator), 976
hpx::error::future_does_not_support_cancellation (C++ enumerator), 975
hpx::error::internal_server_error (C++ enumerator), 974
hpx::error::invalid_data (C++ enumerator), 974
hpx::error::invalid_status (C++ enumerator), 974
hpx::error::kernel_error (C++ enumerator), 975
hpx::error::length_error (C++ enumerator), 976
hpx::error::lock_error (C++ enumerator), 974
hpx::error::migration_needs_retry (C++ enumerator), 976
hpx::error::network_error (C++ enumerator), 973
hpx::error::no_registered_console (C++ enumerator), 974
hpx::error::no_state (C++ enumerator), 975
hpx::error::no_success (C++ enumerator), 973
hpx::error::not_implemented (C++ enumerator), 973
hpx::error::null_thread_id (C++ enumerator), 974
hpx::error::out_of_memory (C++ enumerator), 973
hpx::error::out_of_range (C++ enumerator), 976
hpx::error::promise_already_satisfied (C++ enumerator), 975
hpx::error::repeated_request (C++ enumerator), 974
hpx::error::serialization_error (C++ enumerator), 975
hpx::error::service_unavailable (C++ enumerator), 974
hpx::error::startup_timed_out (C++ enumerator), 974
hpx::error::success (C++ enumerator), 973
hpx::error::task_already_started (C++ enumerator), 975
hpx::error::task_block_not_active (C++ enumerator), 976
hpx::error::task_canceled_exception (C++ enumerator), 976
hpx::error::task_moved (C++ enumerator), 975
hpx::error::thread_cancelled (C++ enumerator), 976
hpx::error::thread_not_interruptable (C++ enumerator), 976
hpx::error::thread_resource_error (C++ enumerator), 975
hpx::error::unhandled_exception (C++ enumerator), 975
hpx::error::uninitialized_value (C++ enumerator), 974
hpx::error::unknown_component_address (C++ enumerator), 974
hpx::error::unknown_error (C++ enumerator), 976
hpx::error::version_too_new (C++ enumerator), 973
hpx::error::version_too_old (C++ enumerator), 973
hpx::error::version_unknown (C++ enumerator), 973
hpx::error::yield_aborted (C++ enumerator), 975
hpx::error_code (C++ class), 977
hpx::error_code::clear (C++ function), 979
hpx::error_code::error_code (C++ function), 978–980
hpx::error_code::exception_ (C++ member), 980
hpx::error_code::get_message (C++ function), 979
hpx::error_code::make_error_code (C++ func-

tion), 980
hpx::error_code::operator=(C++ function), 979
hpx::exception (C++ class), 982
hpx::exception::~exception (C++ function), 983
hpx::exception::exception (C++ function), 983
hpx::exception::get_error (C++ function), 983
hpx::exception::get_error_code (C++ function), 983
hpx::exception_list (C++ class), 986
hpx::exception_list::begin (C++ function), 986
hpx::exception_list::end (C++ function), 986
hpx::exception_list::iterator (C++ type), 986
hpx::exception_list::size (C++ function), 986
hpx::exclusive_scan (C++ function), 319–321
hpx::execution (C++ type), 288, 942, 988, 989, 992, 996, 997, 1001, 1004, 1009, 1014, 1016, 1018, 1021, 1022, 1026, 1027
hpx::execution::adaptive_static_chunk_size (C++ struct), 988
hpx::execution::adaptive_static_chunk_size::adaptive_static_chunk_size (C++ function), 988
hpx::execution::auto_chunk_size (C++ struct), 988
hpx::execution::auto_chunk_size::auto_chunk_size (C++ function), 989
hpx::execution::dynamic_chunk_size (C++ struct), 989
hpx::execution::dynamic_chunk_size::dynamic_chunk_size (C++ function), 989
hpx::execution::experimental (C++ type), 288, 942, 992, 1009, 1014, 1017, 1018, 1021, 1026, 1027
hpx::execution::experimental::annotating_executor (C++ struct), 1014
hpx::execution::experimental::annotating_executor (C++ function), 1014
hpx::execution::experimental::block_fork_join_executor (C++ class), 942
hpx::execution::experimental::block_fork_join_executor (C++ member), 943
hpx::execution::experimental::block_fork_join_executor (C++ function), 942
hpx::execution::experimental::block_fork_join_executor (C++ function), 943
hpx::execution::experimental::block_fork_join_executor (C++ function), 943
hpx::execution::experimental::block_fork_join_executor (C++ member), 943
hpx::execution::experimental::block_fork_join_executor (C++ function), 943, 944
hpx::execution::experimental::explicit_scheduler (C++ function), 1022
hpx::execution::experimental::explicit_scheduler (C++ struct), 1022
hpx::execution::experimental::is_execution_policy_mapping (C++ struct), 1019
hpx::execution::experimental::is_execution_policy_mapping (C++ member), 1018
hpx::execution::experimental::is_execution_policy_mapping (C++ struct), 1017
hpx::execution::experimental::is_execution_policy_mapping (C++ struct), 1017
hpx::execution::experimental::is_execution_policy_mapping (C++ struct), 1019
hpx::execution::experimental::is_nothrow_receiver_of (C++ struct), 1010
hpx::execution::experimental::is_nothrow_receiver_of_v (C++ member), 1010
hpx::execution::experimental::is_receiver (C++ struct), 1010
hpx::execution::experimental::is_receiver_of (C++ struct), 1011
hpx::execution::experimental::is_receiver_of_v (C++ member), 1010
hpx::execution::experimental::is_receiver_v (C++ member), 1010
hpx::execution::experimental::is_scheduling_property<hpx::execution::experimental::scheduler_executor> (C++ struct), 992
hpx::execution::experimental::scheduler_executor (C++ function), 1026
hpx::execution::experimental::scheduler_executor (C++ struct), 1026
hpx::execution::experimental::set_error (C++ function), 1010
hpx::execution::experimental::set_error (C++ member), 1010
hpx::execution::experimental::set_error (C++ function), 1010
hpx::execution::experimental::set_error (C++ member), 1010
hpx::execution::experimental::set_error (C++ function), 1011
hpx::execution::experimental::set_stopped (C++ function), 1009
hpx::execution::experimental::set_stopped (C++ member), 1010
hpx::execution::experimental::set_stopped (C++ member), 1010
hpx::execution::experimental::set_stopped_t (C++ struct), 1011
hpx::execution::experimental::set_value (C++ function), 1009
hpx::execution::experimental::set_value (C++ member), 1010

hpx::execution::experimental::set_value_t (C++ struct), 1011
hpx::execution::experimental::tagFallbackInvoke (C++ function), 1014
hpx::execution::experimental::tagInvoke (C++ function), 1014, 1018, 1022, 1026, 1028
hpx::execution::experimental::task_group (C++ class), 288
hpx::execution::experimental::task_group::~task_group (C++ function), 288
hpx::execution::experimental::task_group::add (C++ function), 288
hpx::execution::experimental::task_group::errd (C++ member), 289
hpx::execution::experimental::task_group::hasArrived (C++ member), 289
hpx::execution::experimental::task_group::latency (C++ member), 289
hpx::execution::experimental::task_group::onExit (C++ struct), 289
hpx::execution::experimental::task_group::onExit::exception (C++ function), 289
hpx::execution::experimental::task_group::onExit::operation (C++ function), 289
hpx::execution::experimental::task_group::run (C++ function), 288
hpx::execution::experimental::task_group::serialize (C++ function), 288
hpx::execution::experimental::task_group::sharedState (C++ type), 288
hpx::execution::experimental::task_group::state (C++ member), 289
hpx::execution::experimental::task_group::task_group (C++ function), 288
hpx::execution::experimental::task_group::waith (C++ function), 288
hpx::execution::experimental::thread_pool_policy (C++ struct), 1028
hpx::execution::experimental::thread_pool_policy::exponentialCategoryTag (C++ type), 1028
hpx::execution::experimental::thread_pool_policy::fixedScheduler (C++ function), 1028
hpx::execution::experimental::thread_pool_scheduler (C++ type), 1028
hpx::execution::experimental::to_non_parallel_executor (C++ member), 1018
hpx::execution::experimental::to_non_parallel_t (C++ struct), 1019
hpx::execution::experimental::to_non_parallel_t::tagFallbackInvoke (C++ function), 1019
hpx::execution::experimental::to_non_task (C++ member), 1018
hpx::execution::experimental::to_non_task_t (C++ struct), 1019
hpx::execution::experimental::to_non_task_t::tagFallbackInvoke (C++ function), 1020
hpx::execution::experimental::to_non_unseq (C++ member), 1019
hpx::execution::experimental::to_non_unseq_t (C++ struct), 1020
hpx::execution::experimental::to_non_unseq_t::tagFallbackInvoke (C++ function), 1020
hpx::execution::experimental::to_parallel (C++ member), 1018
hpx::execution::experimental::to_parallel_t (C++ struct), 1020
hpx::execution::experimental::to_parallel_t::tagFallbackInvoke (C++ function), 1020
hpx::execution::experimental::to_task (C++ member), 1019
hpx::execution::experimental::to_task_t (C++ struct), 1020
hpx::execution::experimental::to_task_t::tagFallbackInvoke (C++ function), 1020
hpx::execution::experimental::to_unseq (C++ member), 1019
hpx::execution::experimental::to_unseq_t (C++ struct), 1020
hpx::execution::experimental::to_unseq_t::tagFallbackInvoke (C++ function), 1020
hpx::execution::experimental::guided_chunk_size (C++ struct), 996
hpx::execution::experimental::guided_chunk_size::guided_chunk_size (C++ function), 996
hpx::execution::non_task (C++ member), 1017
hpx::execution::non_task_policy_tag (C++ struct), 1017
hpx::execution::num_cores (C++ struct), 997
hpx::execution::num_cores::num_cores (C++ function), 997
hpx::execution::par (C++ member), 1017
hpx::execution::par_unseq (C++ member), 1017
hpx::execution::parallel_executor::parallel_executor (C++ type), 1004
hpx::execution::parallel_executor::parallel_executor (C++ type), 1022
hpx::execution::parallel_executor::parallel_executor (C++ type), 1016
hpx::execution::parallel_policy (C++ type), 1016
hpx::execution::parallel_policy_executor (C++ struct), 1023
hpx::execution::parallel_policy_executor::execution_category (C++ type), 1023
hpx::execution::parallel_policy_executor::executor_parameters (C++ type), 1023

hpx::execution::parallel_policy_executor::parallel_filesystem::initial_path (C++ function),
 (C++ function), 1023, 1024
 hpx::execution::parallel_policy_executor::process_infile (C++ function), 323, 324
 (C++ function), 1024
 hpx::execution::parallel_policy_executor::tag::finalize (C++ function), 1325, 1326
 (C++ function), 1024
 hpx::execution::parallel_task_policy (C++ type), 1016
 hpx::execution::parallel_unsequenced_policy
 (C++ type), 1017
 hpx::execution::parallel_unsequenced_task_policy
 (C++ type), 1016
 hpx::execution::persistent_auto_chunk_size
 (C++ struct), 997
 hpx::execution::persistent_auto_chunk_size::persist_and_finalize (C++ function), 328, 329
 (C++ function), 998
 hpx::execution::seq (C++ member), 1017
 hpx::execution::sequenced_execution_tag
 (C++ struct), 1004
 hpx::execution::sequenced_executor
 (C++ struct), 1027
 hpx::execution::sequenced_policy (C++ type),
 1016
 hpx::execution::sequenced_task_policy (C++ type),
 1016
 hpx::execution::static_chunk_size (C++ struct),
 1001
 hpx::execution::static_chunk_size::static_chunk_size (C++ function), 1002
 hpx::execution::tag_invoke (C++ function), 1023
 hpx::execution::task (C++ member), 1017
 hpx::execution::task_policy_tag (C++ struct),
 1017
 hpx::execution::unseq (C++ member), 1017
 hpx::execution::unsequenced_execution_tag
 (C++ struct), 1004
 hpx::execution::unsequenced_policy (C++ type),
 1017
 hpx::execution::unsequenced_task_policy
 (C++ type), 1017
 hpx::experimental (C++ type), 341, 351, 352, 1151
 hpx::experimental::for_loop (C++ function), 341,
 342
 hpx::experimental::for_loop_n (C++ function),
 346, 347
 hpx::experimental::for_loop_n_strided (C++ function), 348, 350
 hpx::experimental::for_loop_strided (C++ function), 344, 345
 hpx::experimental::induction (C++ function), 351
 hpx::experimental::reduction (C++ function), 352
 hpx::filesystem (C++ type), 1029
 hpx::filesystem::basename (C++ function), 1029
 hpx::filesystem::canonical (C++ function), 1029
 1029
 hpx::fill_n (C++ function), 324, 325
 hpx::find (C++ function), 326
 hpx::find_all_from_basename (C++ function), 1296
 hpx::find_all_localities (C++ function), 1383,
 1386
 hpx::find_end (C++ function), 330–333
 hpx::find_first_of (C++ function), 333, 335–337
 hpx::find_from_basename (C++ function), 1297
 hpx::find_here (C++ function), 1385
 hpx::find_if (C++ function), 327, 328
 hpx::find_if_and_finalize (C++ function), 328, 329
 hpx::find_locality (C++ function), 1387
 hpx::find_remote_localities (C++ function),
 1384, 1386
 hpx::find_root_locality (C++ function), 1383
 hpx::for_each (C++ function), 338
 hpx::for_each_n (C++ function), 339, 340
 hpx::forward_as_tuple (C++ function), 967
 hpx::function (C++ class), 1033
 hpx::function_ref (C++ class), 1034
 hpx::function_ref<R(Ts...)> (C++ class), 1035
 hpx::function_ref<R(Ts...)>::assign (C++ function), 1035
 hpx::function_ref<R(Ts...)>::function_ref (C++ function), 1035
 hpx::function_ref<R(Ts...)>::get_function_address (C++ function), 1035
 hpx::function_ref<R(Ts...)>::get_function_annotation (C++ function), 1035
 hpx::function_ref<R(Ts...)>::get_function_annotation_itt (C++ function), 1035
 hpx::function_ref<R(Ts...)>::get_vtable (C++ function), 1036
 hpx::function_ref<R(Ts...)>::object (C++ member), 1036
 hpx::function_ref<R(Ts...)>::operator() (C++ function), 1035
 hpx::function_ref<R(Ts...)>::operator= (C++ function), 1035
 hpx::function_ref<R(Ts...)>::swap (C++ function), 1035
 hpx::function_ref<R(Ts...)>::vptr (C++ member), 1036
 hpx::function_ref<R(Ts...)>::VTable (C++ type), 1036
 hpx::function<R(Ts...), Serializable> (C++ class), 1033
 hpx::function<R(Ts...), Serializable>::base_type (C++ type),
 1034

hpx::function<R(Ts...),
 Serializable>::function (C++ function), 1033, 1034
hpx::function<R(Ts...),
 Serializable>::operator= (C++ function), 1033, 1034
hpx::function<R(Ts...),
 Serializable>::result_type (C++ type), 1033
hpx::functional (C++ type), 1037, 1069
hpx::functional::invoke (C++ struct), 1037
hpx::functional::invoke::operator() (C++ function), 1037
hpx::functional::invoke_r (C++ struct), 1037
hpx::functional::invoke_r::operator() (C++ function), 1037
hpx::functional::unwrap (C++ struct), 1069
hpx::functional::unwrap_all (C++ struct), 1069
hpx::functional::unwrap_n (C++ struct), 1069
hpx::future (C++ class), 1045, 1050
hpx::future::~future (C++ function), 1046
hpx::future::base_type (C++ type), 1046
hpx::future::future (C++ function), 1046, 1047
hpx::future::get (C++ function), 1046
hpx::future::invalidate (C++ struct), 1047
hpx::future::invalidate::~invalidate (C++ function), 1047
hpx::future::invalidate::f_ (C++ member), 1047
hpx::future::invalidate::invalidate (C++ function), 1047
hpx::future::operator= (C++ function), 1046
hpx::future::result_type (C++ type), 1046
hpx::future::share (C++ function), 1046
hpx::future::shared_state_type (C++ type), 1046
hpx::future::then (C++ function), 1046
hpx::future::then_alloc (C++ function), 1046
hpx::generate (C++ function), 353, 354
hpx::generate_n (C++ function), 355
hpx::get (C++ function), 967
hpx::get_colocation_id (C++ function), 1237
hpx::get_error (C++ function), 981
hpx::get_error_backtrace (C++ function), 1084
hpx::get_error_config (C++ function), 1086
hpx::get_error_env (C++ function), 1083
hpx::get_error_file_name (C++ function), 982
hpx::get_error_function_name (C++ function), 981
hpx::get_error_host_name (C++ function), 1082
hpx::get_error_line_number (C++ function), 982
hpx::get_error_locality_id (C++ function), 1082
hpx::get_error_os_thread (C++ function), 1084
hpx::get_error_process_id (C++ function), 1083
hpx::get_error_state (C++ function), 1086
hpx::get_error_thread_description (C++ function), 1085
hpx::get_error_thread_id (C++ function), 1085
hpx::get_error_what (C++ function), 980
hpx::get_hpx_category (C++ function), 977
hpx::get_hpx_rethrow_category (C++ function), 977
hpx::get_initial_num_localities (C++ function), 1088
hpx::get_local_worker_thread_num (C++ function), 1184
hpx::get_locality_id (C++ function), 1087
hpx::get_locality_name (C++ function), 1087, 1388
hpx::get_lva (C++ struct), 1312
hpx::get_lva::call (C++ function), 1312
hpx::get_lva<lcos::base_lco const> (C++ struct), 1238
hpx::get_lva<lcos::base_lco const>::call (C++ function), 1238
hpx::get_lva<lcos::base_lco> (C++ struct), 1238
hpx::get_lva<lcos::base_lco>::call (C++ function), 1238
hpx::get_num_localities (C++ function), 1088, 1388, 1389
hpx::get_num_worker_threads (C++ function), 1099
hpx::get_os_thread_count (C++ function), 1089
hpx::get_os_thread_data (C++ function), 1098
hpx::get_ptr (C++ function), 1301, 1302
hpx::get_runtime_instance_number (C++ function), 1098
hpx::get_runtime_mode_from_name (C++ function), 1080
hpx::get_runtime_mode_name (C++ function), 1080
hpx::get_runtime_support_ptr (C++ function), 1367
hpx::get_system_uptime (C++ function), 1099
hpx::get_thread_name (C++ function), 1089
hpx::get_thread_on_error_func (C++ function), 1104
hpx::get_thread_on_start_func (C++ function), 1104
hpx::get_thread_on_stop_func (C++ function), 1104
hpx::get_thread_pool_num (C++ function), 1184, 1185
hpx::get_worker_thread_num (C++ function), 1183
hpx::identity (C++ type), 881
hpx::ignore (C++ member), 967
hpx::includes (C++ function), 356, 357
hpx::inclusive_scan (C++ function), 359–361, 363, 364
hpx::init (C++ function), 1328–1330
hpx::init_params (C++ struct), 1331
hpx::init_params::cfg (C++ member), 1331
hpx::init_params::desc_cmdline (C++ member), 1331

hpx::init_params::init_params (*C++ function*), 1331
 hpx::init_params::mode (*C++ member*), 1331
 hpx::init_params::rp_callback (*C++ member*), 1332
 hpx::init_params::rp_mode (*C++ member*), 1331
 hpx::init_params::shutdown (*C++ member*), 1331
 hpx::init_params::startup (*C++ member*), 1331
 hpx::inplace_merge (*C++ function*), 381, 382
 hpx::invoke (*C++ function*), 1036
 hpx::invoke_fused (*C++ function*), 1038
 hpx::invoke_fused_r (*C++ function*), 1038
 hpx::invoke_r (*C++ function*), 1037
 hpx::is_async_execution_policy (*C++ struct*), 1002
 hpx::is_async_execution_policy_v (*C++ member*), 1002
 hpx::is_bind_expression (*C++ struct*), 1042
 hpx::is_bind_expression_v (*C++ member*), 1042
 hpx::is_bind_expression<T const> (*C++ struct*), 1042
 hpx::is_execution_policy (*C++ struct*), 1003
 hpx::is_execution_policy_v (*C++ member*), 1002
 hpx::is_heap (*C++ function*), 365, 366
 hpx::is_heap_until (*C++ function*), 367
 hpx::is_invocable (*C++ struct*), 1154
 hpx::is_invocable_r (*C++ struct*), 1155
 hpx::is_invocable_r_v (*C++ member*), 1154
 hpx::is_invocable_v (*C++ member*), 1154
 hpx::is_nothrow_invocable (*C++ struct*), 1155
 hpx::is_nothrow_invocable_v (*C++ member*), 1154
 hpx::is_parallel_execution_policy (*C++ struct*), 1003
 hpx::is_parallel_execution_policy_v (*C++ member*), 1002
 hpx::is_partitioned (*C++ function*), 368, 369
 hpx::is_placeholder (*C++ struct*), 1043
 hpx::is_running (*C++ function*), 1099
 hpx::is_sequenced_execution_policy (*C++ struct*), 1003
 hpx::is_sequenced_execution_policy_v (*C++ member*), 1002
 hpx::is_sorted (*C++ function*), 370, 371
 hpx::is_sorted_until (*C++ function*), 372
 hpx::is_starting (*C++ function*), 1099
 hpx::is_stopped (*C++ function*), 1099
 hpx::is_stopped_or_shutting_down (*C++ function*), 1099
 hpx::jthread (*C++ class*), 1158
 hpx::jthread::~jthread (*C++ function*), 1158
 hpx::jthread::detach (*C++ function*), 1159
 hpx::jthread::get_id (*C++ function*), 1159
 hpx::jthread::get_stop_source (*C++ function*), 1159
 hpx::jthread::get_stop_token (*C++ function*), 1159
 hpx::jthread::hardware_concurrency (*C++ function*), 1159
 hpx::jthread::id (*C++ type*), 1158
 hpx::jthread::invoke (*C++ function*), 1160
 hpx::jthread::join (*C++ function*), 1159
 hpx::jthread::joinable (*C++ function*), 1159
 hpx::jthread::jthread (*C++ function*), 1158
 hpx::jthread::native_handle (*C++ function*), 1159
 hpx::jthread::native_handle_type (*C++ type*), 1158
 hpx::jthread::operator= (*C++ function*), 1159
 hpx::jthread::request_stop (*C++ function*), 1159
 hpx::jthread::ssource_ (*C++ member*), 1159
 hpx::jthread::swap (*C++ function*), 1159
 hpx::jthread::thread_ (*C++ member*), 1159
 hpx::latch (*C++ class*), 1133
 hpx::latch::~latch (*C++ function*), 1133
 hpx::latch::arrive_and_wait (*C++ function*), 1133
 hpx::latch::cond_ (*C++ member*), 1134
 hpx::latch::count_down (*C++ function*), 1133
 hpx::latch::counter_ (*C++ member*), 1134
 hpx::latch::HPX_NON_COPYABLE (*C++ function*), 1133
 hpx::latch::latch (*C++ function*), 1133
 hpx::latch::mtx_ (*C++ member*), 1134
 hpx::latch::mutex_type (*C++ type*), 1134
 hpx::latch::notified_ (*C++ member*), 1134
 hpx::latch::try_wait (*C++ function*), 1133
 hpx::latch::wait (*C++ function*), 1133
 hpx::launch (*C++ struct*), 889
 hpx::launch::apply (*C++ member*), 890
 hpx::launch::async (*C++ member*), 890
 hpx::launch::deferred (*C++ member*), 890
 hpx::launch::fork (*C++ member*), 890
 hpx::launch::launch (*C++ function*), 890
 hpx::launch::select (*C++ member*), 890
 hpx::launch::sync (*C++ member*), 890
 hpx::launch::tag_invoke (*C++ function*), 891
 hpx::lcos (*C++ type*), 1049, 1050, 1052, 1055, 1058, 1061, 1062, 1126, 1131, 1134, 1141, 1142, 1145–1148, 1238, 1241, 1244, 1245, 1275, 1278, 1281, 1289, 1292
 hpx::lcos::base_lco (*C++ class*), 1238
 hpx::lcos::base_lco::~base_lco (*C++ function*), 1239
 hpx::lcos::base_lco::base_type_holder (*C++ type*), 1239
 hpx::lcos::base_lco::connect (*C++ function*), 1239
 hpx::lcos::base_lco::connect_nonvirt (*C++ function*), 1239

hpx::lcos::base_lco::disconnect (*C++ function*),
1239
hpx::lcos::base_lco::disconnect_nonvirt
(*C++ function*), 1239
hpx::lcos::base_lco::disconnect_nonvirt
(*C++ member*), 1240
hpx::lcos::base_lco::finalize (*C++ function*),
1239
hpx::lcos::base_lco::get_component_type
(*C++ function*), 1240
hpx::lcos::base_lco::set_component_type
(*C++ function*), 1240
hpx::lcos::base_lco::set_event (*C++ function*),
1239
hpx::lcos::base_lco::set_event_nonvirt (*C++
function*), 1239
hpx::lcos::base_lco::set_exception (*C++ function*),
1239
hpx::lcos::base_lco::set_exception_nonvirt
(*C++ function*), 1239
hpx::lcos::base_lco::wrapping_type (*C++ type*),
1239
hpx::lcos::base_lco_with_value (*C++ class*),
1241, 1244
hpx::lcos::base_lco_with_value::~base_lco_with_value
(*C++ function*), 1242
hpx::lcos::base_lco_with_value::base_type_holder
(*C++ type*), 1241
hpx::lcos::base_lco_with_value::get_component_type
(*C++ function*), 1242
hpx::lcos::base_lco_with_value::get_value
(*C++ function*), 1242
hpx::lcos::base_lco_with_value::get_value_nonvirt
(*C++ function*), 1241
hpx::lcos::base_lco_with_value::result_type
(*C++ type*), 1242
hpx::lcos::base_lco_with_value::set_component_type
(*C++ function*), 1242
hpx::lcos::base_lco_with_value::set_event
(*C++ function*), 1242
hpx::lcos::base_lco_with_value::set_event_nonvirt
(*C++ function*), 1242
hpx::lcos::base_lco_with_value::set_value
(*C++ function*), 1242
hpx::lcos::base_lco_with_value::set_value_nonvirt
(*C++ function*), 1241
hpx::lcos::base_lco_with_value::wrapping_type
(*C++ type*), 1241
hpx::lcos::base_lco_with_value<void, void,
ComponentTag> (*C++ class*), 1242
hpx::lcos::base_lco_with_value<void, void,
ComponentTag>::~base_lco_with_value
(*C++ function*), 1243
hpx::lcos::base_lco_with_value<void, void,
ComponentTag>::base_type_holder (*C++
type*), 1243
hpx::lcos::base_lco_with_value<void, void,
ComponentTag>::get_value (*C++ func-
tion*), 1243
hpx::lcos::base_lco_with_value<void, void,
ComponentTag>::set_value_action (*C++
type*), 1243
hpx::lcos::base_lco_with_value<void, void,
ComponentTag>::wrapping_type (*C++
type*), 1243
hpx::lcos::create_communication_set
(*C++
function*), 1278
hpx::lcos::instead (*C++ type*), 1050, 1244
hpx::lcos::local (*C++ type*), 1052, 1055, 1058,
1061, 1062, 1126, 1131, 1134, 1141, 1142,
1145–1148
hpx::lcos::local::and_gate (*C++ struct*), 1058
hpx::lcos::local::and_gate::and_gate
(*C++
function*), 1058
hpx::lcos::local::and_gate::base_type
(*C++
type*), 1059
hpx::lcos::local::and_gate::get_future
(*C++
function*), 1058
hpx::lcos::local::and_gate::operator=
(*C++
function*), 1058
hpx::lcos::local::and_gate::set (*C++ function*),
1058
hpx::lcos::local::and_gate::synchronize
(*C++function*), 1058
hpx::lcos::local::base_and_gate (*C++ struct*),
1059
hpx::lcos::local::base_and_gate::base_and_gate
(*C++function*), 1059
hpx::lcos::local::base_and_gate::condition_list_type
(*C++ type*), 1060
hpx::lcos::local::base_and_gate::conditions_
(*C++ member*), 1060
hpx::lcos::local::base_and_gate::generation
(*C++function*), 1059
hpx::lcos::local::base_and_gate::generation_
(*C++ member*), 1060
hpx::lcos::local::base_and_gate::get_future
(*C++function*), 1059, 1060
hpx::lcos::local::base_and_gate::get_shared_future
(*C++function*), 1059, 1060
hpx::lcos::local::base_and_gate::init_locked
(*C++function*), 1060
hpx::lcos::local::base_and_gate::manage_condition
(*C++ struct*), 1060
hpx::lcos::local::base_and_gate::manage_condition::~manage
(*C++function*), 1061

```

hpx::lcos::local::base_and_gate::manage_condition hpx::lcos::local::base_trigger::mtx_      (C++  

(C++ function), 1061                               member), 1063
hpx::lcos::local::base_and_gate::manage_condition hpx::lcos::local::base_trigger::mutex_type  

(C++ member), 1061                               (C++ type), 1063
hpx::lcos::local::base_and_gate::manage_condition hpx::lcos::local::base_trigger::next_generation  

(C++ function), 1061                               (C++ function), 1062
hpx::lcos::local::base_and_gate::manage_condition hpx::lcos::local::base_trigger::operator=  

(C++ member), 1061                               (C++ function), 1062
hpx::lcos::local::base_and_gate::mtx_   (C++ hpx::lcos::local::base_trigger::promise_  

member), 1060                               (C++ member), 1063
hpx::lcos::local::base_and_gate::mutex_type    hpx::lcos::local::base_trigger::set      (C++  

(C++ type), 1059                               function), 1062
hpx::lcos::local::base_and_gate::next_generation hpx::lcos::local::base_trigger::synchronize  

(C++ function), 1059                               (C++ function), 1062, 1063
hpx::lcos::local::base_and_gate::operator=     hpx::lcos::local::base_trigger::test_condition  

(C++ function), 1059                               (C++ function), 1063
hpx::lcos::local::base_and_gate::promise_     hpx::lcos::local::base_trigger::trigger_conditions  

(C++ member), 1060                               (C++ function), 1063
hpx::lcos::local::base_and_gate::received_segments hpx::lcos::local::conditional_trigger  (C++  

(C++ member), 1060                               struct), 1061
hpx::lcos::local::base_and_gate::set      (C++ hpx::lcos::local::conditional_trigger::cond_  

function), 1059, 1060                               (C++ member), 1062
hpx::lcos::local::base_and_gate::synchronize  hpx::lcos::local::conditional_trigger::conditional_trigger  

(C++ function), 1059, 1060                               (C++ function), 1061
hpx::lcos::local::base_and_gate::test_condition hpx::lcos::local::conditional_trigger::get_future  

(C++ function), 1060                               (C++ function), 1061
hpx::lcos::local::base_and_gate::trigger_conditions hpx::lcos::local::conditional_trigger::operator=  

(C++ function), 1060                               (C++ function), 1061
hpx::lcos::local::base_trigger  (C++ struct),  hpx::lcos::local::conditional_trigger::promise_  

1062                                         (C++ member), 1062
hpx::lcos::local::base_trigger::base_trigger  hpx::lcos::local::conditional_trigger::reset  

(C++ function), 1062                               (C++ function), 1061
hpx::lcos::local::base_trigger::condition_list hpx::lcos::local::conditional_trigger::set  

(C++ type), 1063                               (C++ function), 1061
hpx::lcos::local::base_trigger::conditions_  hpx::lcos::local::event (C++ class), 1131
(C++ member), 1063                               hpx::lcos::local::event::cond_ (C++ member),  

1132
hpx::lcos::local::base_trigger::generation  hpx::lcos::local::event::event (C++ function),  

(C++ function), 1062                               1132
hpx::lcos::local::base_trigger::generation_ hpx::lcos::local::event::event_ (C++ member),  

(C++ member), 1063                               1132
hpx::lcos::local::base_trigger::get_future   hpx::lcos::local::event::mtx_ (C++ member),  

(C++ function), 1062                               1132
hpx::lcos::local::base_trigger::manage_condition hpx::lcos::local::event::mutex_type      (C++  

(C++ struct), 1063                               member), 1132
hpx::lcos::local::base_trigger::manage_condition::~manage_condition  

(C++ function), 1063                               hpx::lcos::local::event::occurred (C++ func-
hpx::lcos::local::base_trigger::manage_condition::get_future 1132  

(C++ function), 1063                               hpx::lcos::local::event::reset (C++ function),
hpx::lcos::local::base_trigger::manage_condition::it_ 1132  

(C++ member), 1064                               hpx::lcos::local::event::set (C++ function),
hpx::lcos::local::base_trigger::manage_condition::manage2condition  

(C++ function), 1063                               hpx::lcos::local::event::set_locked      (C++  

hpx::lcos::local::base_trigger::manage_condition::this$function), 1132  

(C++ member), 1064                               hpx::lcos::local::event::wait (C++ function),

```

1132
hpx::lcos::local::event::wait_locked (C++ function), 1132
hpx::lcos::local::instead (C++ type), 1052, 1126
hpx::lcos::local::latch (C++ class), 1134
hpx::lcos::local::latch::~latch (C++ function), 1135
hpx::lcos::local::latch::abort_all (C++ function), 1135
hpx::lcos::local::latch::count_down_and_wait (C++ function), 1135
hpx::lcos::local::latch::count_up (C++ function), 1135
hpx::lcos::local::latch::HPX_NON_COPYABLE (C++ function), 1135
hpx::lcos::local::latch::is_ready (C++ function), 1135
hpx::lcos::local::latch::latch (C++ function), 1135
hpx::lcos::local::latch::reset (C++ function), 1135
hpx::lcos::local::latch::reset_if_needed_and_count_up (C++ function), 1135
hpx::lcos::local::trigger (C++ struct), 1064
hpx::lcos::local::trigger::base_type (C++ type), 1064
hpx::lcos::local::trigger::operator= (C++ function), 1064
hpx::lcos::local::trigger::synchronize (C++ function), 1064
hpx::lcos::local::trigger::trigger (C++ function), 1064
hpx::lcos::object_semaphore (C++ struct), 1244
hpx::lcos::packaged_action (C++ class), 1244, 1245
hpx::lcos::packaged_action<Action, Result, false> (C++ class), 1245
hpx::lcos::packaged_action<Action, Result, false>::action_type (C++ type), 1246
hpx::lcos::packaged_action<Action, Result, false>::apply (C++ function), 1245
hpx::lcos::packaged_action<Action, Result, false>::apply_cb (C++ function), 1245
hpx::lcos::packaged_action<Action, Result, false>::apply_deferred (C++ function), 1246
hpx::lcos::packaged_action<Action, Result, false>::apply_deferred_cb (C++ function), 1246
hpx::lcos::packaged_action<Action, Result, false>::apply_p (C++ function), 1246
hpx::lcos::packaged_action<Action, Result, false>::apply_p_cb (C++ function), 1246
hpx::lcos::packaged_action<Action, Result, false>::base_type (C++ type), 1246
hpx::lcos::packaged_action<Action, Result, false>::do_apply (C++ function), 1246
hpx::lcos::packaged_action<Action, Result, false>::do_apply_cb (C++ function), 1246
hpx::lcos::packaged_action<Action, Result, false>::packaged_action (C++ function), 1245
hpx::lcos::packaged_action<Action, Result, false>::remote_result_type (C++ type), 1246
hpx::lcos::packaged_action<Action, Result, true> (C++ class), 1246
hpx::lcos::packaged_action<Action, Result, true>::action_type (C++ type), 1247
hpx::lcos::packaged_action<Action, Result, true>::apply (C++ function), 1247
hpx::lcos::packaged_action<Action, Result, true>::apply_cb (C++ function), 1247
hpx::lcos::packaged_action<Action, Result, true>::packaged_action (C++ function), 1247
hpx::lcos::server (C++ type), 1245
hpx::lcos::server::object_semaphore (C++ struct), 1245
hpx::lexicographical_compare (C++ function), 374, 375
hpx::lightweight (C++ member), 985
hpx::lightweight_rethrow (C++ member), 985
hpx::make_any (C++ function), 970
hpx::make_any_nonsr (C++ function), 959
hpx::make_error_code (C++ function), 977
hpx::make_exceptional_future (C++ function), 1045
hpx::make_future (C++ function), 1044
hpx::make_heap (C++ function), 376–379
hpx::make_ready_future (C++ function), 1044, 1045
hpx::make_ready_future_after (C++ function), 1045
hpx::make_ready_future_alloc (C++ function), 1044, 1045
hpx::make_ready_future_at (C++ function), 1045
hpx::make_shared_future (C++ function), 1044
hpx::make_success_code (C++ function), 977
hpx::make_tuple (C++ function), 967
hpx::make_unique_any_nonsr (C++ function), 959
hpx::max_element (C++ function), 385, 655–657
hpx::mem_fn (C++ function), 1039, 1040
hpx::merge (C++ function), 379, 380
hpx::min_element (C++ function), 383, 384, 651–653
hpx::minmax_element (C++ function), 386, 387, 659, 660, 662
hpx::mismatch (C++ function), 389–391, 393–396
hpx::move (C++ function), 397, 398

hpx::move_only_function (C++ class), 1040
 hpx::move_only_function<R(Ts...),
 Serializable> (C++ class), 1041
 hpx::move_only_function<R(Ts...),
 Serializable>::base_type (C++ type),
 1041
 hpx::move_only_function<R(Ts...),
 Serializable>::move_only_function
 (C++ function), 1041
 hpx::move_only_function<R(Ts...),
 Serializable>::operator= (C++ function), 1041
 hpx::move_only_function<R(Ts...),
 Serializable>::result_type (C++ type), 1041
 hpx::mpi (C++ type), 912
 hpx::mpi::experimental (C++ type), 912
 hpx::mpi::experimental::executor (C++ struct),
 912
 hpx::mpi::experimental::executor::communicator_
 (C++ member), 912
 hpx::mpi::experimental::executor::execution_category
 (C++ type), 912
 hpx::mpi::experimental::executor::executor
 (C++ function), 912
 hpx::mpi::experimental::executor::executor
 (C++ type), 912
 hpx::mpi::experimental::executor::in_flight_estimate
 (C++ function), 912
 hpx::mpi::experimental::executor::tag_invoke
 (C++ function), 912
 hpx::mpi::experimental::transform_mpi (C++ member), 913
 hpx::mpi::experimental::transform_mpi_t
 (C++ struct), 913
 hpx::mutex (C++ class), 1136
 hpx::mutex::~mutex (C++ function), 1136
 hpx::mutex::HPX_NON_COPYABLE (C++ function),
 1136
 hpx::mutex::lock (C++ function), 1136, 1137
 hpx::mutex::mutex (C++ function), 1136
 hpx::mutex::try_lock (C++ function), 1137
 hpx::mutex::unlock (C++ function), 1138
 hpx::naming (C++ type), 1310, 1336
 hpx::naming::operator<< (C++ function), 1311
 hpx::naming::unmanaged (C++ function), 1336
 hpx::no_mutex (C++ struct), 1142
 hpx::no_mutex::lock (C++ function), 1142
 hpx::no_mutex::try_lock (C++ function), 1142
 hpx::no_mutex::unlock (C++ function), 1142
 hpx::none_of (C++ function), 295, 296
 hpx::nostopstate (C++ member), 1149
 hpx::nostopstate_t (C++ struct), 1149
 hpx::nostopstate_t::nostopstate_t (C++ func-
 tion), 1149
 hpx::nth_element (C++ function), 399
 hpx::once_flag (C++ struct), 1143
 hpx::once_flag::call_once (C++ function), 1144
 hpx::once_flag::event_ (C++ member), 1144
 hpx::once_flag::HPX_NON_COPYABLE (C++ func-
 tion), 1144
 hpx::once_flag::once_flag (C++ function), 1144
 hpx::once_flag::status_ (C++ member), 1144
 hpx::operator!= (C++ function), 1160
 hpx::operator== (C++ function), 1160
 hpx::operator& (C++ function), 985
 hpx::operator> (C++ function), 1160
 hpx::operator>= (C++ function), 1160
 hpx::operator< (C++ function), 1160
 hpx::operator<= (C++ function), 1160
 hpx::operator<< (C++ function), 886, 1160
 hpx::p2300_stop_token (C++ type), 1151
 hpx::p2300_stop_token::in_place_stop_callback
 (C++ class), 1152
 hpx::p2300_stop_token::in_place_stop_callback
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source
 (C++ class), 1152
 hpx::p2300_stop_token::in_place_stop_source::~in_place_sto
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::get_token
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::in_place_stop
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::operator=
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::register_call
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::remove_callba
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::request_stop
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::state_
 (C++ member), 1152
 hpx::p2300_stop_token::in_place_stop_source::stop_possible
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_source::stop_requeste
 (C++ function), 1152
 hpx::p2300_stop_token::in_place_stop_token
 (C++ class), 1152
 hpx::p2300_stop_token::in_place_stop_token::callback_type
 (C++ type), 1153
 hpx::p2300_stop_token::in_place_stop_token::in_place_stop_
 (C++ function), 1153
 hpx::p2300_stop_token::in_place_stop_token::operator=
 (C++ function), 1153
 hpx::p2300_stop_token::in_place_stop_token::source_
 (C++ member), 1153

hpx::p2300_stop_token::in_place_stop_token::stop_possible (C++ function), 1153
hpx::p2300_stop_token::in_place_stop_token::stop_request (C++ function), 1153
hpx::p2300_stop_token::in_place_stop_token::swap (C++ function), 1153
hpx::p2300_stop_token::never_stop_token (C++ struct), 1153
hpx::p2300_stop_token::never_stop_token::callback_impl (C++ struct), 1192
hpx::p2300_stop_token::never_stop_token::callback_type (C++ type), 1154
hpx::p2300_stop_token::never_stop_token::stop_possible (C++ function), 1154
hpx::p2300_stop_token::never_stop_token::stop_request (C++ function), 1154
hpx::packaged_task (C++ class), 1051
hpx::packaged_task<R(Ts...)> (C++ class), 1051
hpx::packaged_task<R(Ts...)>::function_ (C++ member), 1052
hpx::packaged_task<R(Ts...)>::function_type (C++ type), 1052
hpx::packaged_task<R(Ts...)>::get_future (C++ function), 1051
hpx::packaged_task<R(Ts...)>::operator() (C++ function), 1051
hpx::packaged_task<R(Ts...)>::operator= (C++ function), 1051
hpx::packaged_task<R(Ts...)>::packaged_task (C++ function), 1051
hpx::packaged_task<R(Ts...)>::promise_ (C++ member), 1052
hpx::packaged_task<R(Ts...)>::reset (C++ function), 1051
hpx::packaged_task<R(Ts...)>::set_exception (C++ function), 1051
hpx::packaged_task<R(Ts...)>::swap (C++ function), 1051
hpx::packaged_task<R(Ts...)>::valid (C++ function), 1051
hpx::parallel (C++ type), 283, 352, 353, 417, 461, 875, 878–881, 911, 912, 941, 944, 988–990, 992, 993, 996–998, 1000, 1002, 1004, 1012, 1014–1016, 1021, 1022, 1024–1027, 1072, 1074, 1100, 1191, 1195, 1196, 1324, 1401–1409, 1412
hpx::parallel::define_task_block (C++ function), 283, 284
hpx::parallel::define_task_block_restore_thread (C++ function), 284
hpx::parallel::execution (C++ type), 911, 912, 941, 944, 988–990, 992, 993, 996–998, 1000, 1004, 1012, 1021, 1022, 1024–1027, 1072, 1074, 1100, 1191, 1195, 1196, 1324, 1401–1409, 1412
hpx::parallel::execution::async_execute (C++ member), 1004
hpx::parallel::execution::async_execute_after (C++ member), 1192
hpx::parallel::execution::async_execute_after_t (C++ struct), 1192
hpx::parallel::execution::async_execute_at (C++ member), 1192
hpx::parallel::execution::async_execute_at_t (C++ struct), 1192
hpx::parallel::execution::bulk_async_execute (C++ member), 1005
hpx::parallel::execution::bulk_async_execute_t (C++ struct), 1005
hpx::parallel::execution::bulk_async_execute_t::tag_fallback_in (C++ function), 1005
hpx::parallel::execution::bulk_sync_execute (C++ member), 1005
hpx::parallel::execution::bulk_sync_execute_t (C++ struct), 1006
hpx::parallel::execution::bulk_sync_execute_t::tag_fallback_in (C++ function), 1006
hpx::parallel::execution::bulk_then_execute (C++ member), 1005
hpx::parallel::execution::bulk_then_execute_t (C++ struct), 1007
hpx::parallel::execution::bulk_then_execute_t::tag_fallback_in (C++ function), 1007
hpx::parallel::execution::create_rebound_policy (C++ member), 1001
hpx::parallel::execution::create_rebound_policy_t (C++ struct), 1001
hpx::parallel::execution::create_rebound_policy_t::operator() (C++ function), 1001
hpx::parallel::execution::distribution_policy_executor (C++ class), 1324
hpx::parallel::execution::distribution_policy_executor::distribution (C++ function), 1324
hpx::parallel::execution::distribution_policy_executor::policy (C++ member), 1325
hpx::parallel::execution::executor_execution_category<comp> (C++ struct), 939
hpx::parallel::execution::executor_execution_category<comp>

(C++ type), 939
`hpx::parallel::execution::executor_parameters`
 ↳ `join::parallel::execution::has_pending_closures_t::tag_invoke`
 (C++ struct), 992
`hpx::parallel::execution::executor_parameters`
 ↳ `join::parallel::execution::instead` (C++ type),
 (C++ type), 992
`hpx::parallel::execution::executor_parameters`
 ↳ `join::parallel::execution::io_pool_executor`
 (C++ struct), 992
`hpx::parallel::execution::executor_parameters`
 ↳ `join::parallel::execution::io_pool_executor::io_pool_executor`
 (C++ type), 992
`hpx::parallel::execution::extract_executor_parameters`
 ↳ `join::parallel::execution::is_bulk_one_way_executor<compute::`
 (C++ struct), 1012
`hpx::parallel::execution::extract_executor_parameters`
 ↳ `join::parallel::execution::is_bulk_two_way_executor<compute::`
 (C++ type), 1013
`hpx::parallel::execution::extract_executor_parameters`
 ↳ `join::parallel::execution::is_bulk_two_way_executor<hpx::re`
 (C++ type), 1012
`hpx::parallel::execution::extract_executor_parameters`
 ↳ `join::parallel::execution::is_bulk_two_way_executor<hpx::re`
 std::void_t<typename
 Voter, Validator>> (C++ struct), 1074
 Executor::executor_parameters_type>>
 hpx::parallel::execution::is_executor_parameters
 (C++ struct), 1011
`hpx::parallel::execution::extract_executor_parameters`
 ↳ `join::parallel::execution::is_executor_parameters_t`
 std::void_t<typename
 (C++ type), 1012
 Executor::executor_parameters_type>>::
 hpx::parallel::execution::is_executor_parameters_v
 (C++ type), 1012
`hpx::parallel::execution::extract_has_variable`
 ↳ `join::parallel::execution::is_one_way_executor<compute::hos`
 (C++ struct), 1013
`hpx::parallel::execution::extract_has_variable`
 ↳ `join::parallel::execution::is_timed_executor`
 (C++ member), 1012
`hpx::parallel::execution::extract_has_variable`
 ↳ `join::parallel::Parameters::is_timed_executor_t`
 std::void_t<typename
 (C++ type), 1196
 Parameters::has_variable_chunk_size>>
 hpx::parallel::execution::is_timed_executor_v
 (C++ struct), 1012
`hpx::parallel::execution::extract_invokes_testing`
 ↳ `join::parallel::execution::is_two_way_executor<compute::hos`
 (C++ struct), 1013
`hpx::parallel::execution::extract_invokes_testing`
 ↳ `join::parallel::execution::is_two_way_executor<hpx::resilie`
 (C++ member), 1012
`hpx::parallel::execution::get_chunk_size`
 ↳ `join::parallel::execution::is_two_way_executor<hpx::resilie`
 (C++ member), 993
`hpx::parallel::execution::get_chunk_size_t`
 ↳ `join::parallel::execution::join_executor_parameters`
 (C++ function), 992
`hpx::parallel::execution::get_chunk_size_t::tag_if_fallback`
 ↳ `join::parallel::execution::main_pool_executor`
 (C++ function), 994
`hpx::parallel::execution::get_pu_mask`
 ↳ `join::parallel::execution::main_pool_executor::main_pool_ex`
 (C++ member), 990
`hpx::parallel::execution::get_pu_mask_t`
 ↳ `join::parallel::execution::make_distribution_policy_execut`
 (C++ struct), 990
`hpx::parallel::execution::get_pu_mask_t::tag_if_fallback`
 ↳ `join::parallel::execution::mark_begin_execution`
 (C++ function), 991
`hpx::parallel::execution::get_pu_mask_t::tag_if_fallback`
 ↳ `join::parallel::execution::mark_begin_execution_t`
 (C++ function), 991
`hpx::parallel::execution::has_pending_closures`
 ↳ `join::parallel::execution::mark_begin_execution_t::tag_fall`
 (C++ member), 990
`hpx::parallel::execution::has_pending_closures`
 ↳ `join::parallel::execution::mark_end_execution`
 (C++ struct), 991
`hpx::parallel::execution::has_pending_closures`
 ↳ `join::parallel::execution::mark_end_execution_t`
 (C++ member), 993

(*C++ function*), 1101
hpx::parallel::execution::service_executor_type (*C++ enum*), 1100
hpx::parallel::execution::service_executor_type (*C++ enumerator*), 1100
hpx::parallel::execution::service_executor_type (*C++ enumerator*), 1100
hpx::parallel::execution::service_executor_type (*C++ enumerator*), 1100
hpx::parallel::execution::service_executor_type (*C++ enumerator*), 1100
hpx::parallel::execution::set_scheduler_mode (*C++ member*), 990
hpx::parallel::execution::set_scheduler_mode_t (*C++ struct*), 991
hpx::parallel::execution::set_scheduler_mode_t (*C++ function*), 991
hpx::parallel::execution::sync_execute (*C++ member*), 1004
hpx::parallel::execution::sync_execute_after (*C++ member*), 1192
hpx::parallel::execution::sync_execute_after_t (*C++ struct*), 1194
hpx::parallel::execution::sync_execute_at (*C++ member*), 1192
hpx::parallel::execution::sync_execute_at_t (*C++ struct*), 1194
hpx::parallel::execution::sync_execute_at_t::tag_fallback_invoke (*C++ function*), 1195
hpx::parallel::execution::sync_execute_t (*C++ struct*), 1008
hpx::parallel::execution::sync_execute_t::tag_fallback_invoke (*C++ function*), 1008
hpx::parallel::execution::tag_fallback_invoke (*C++ function*), 1021
hpx::parallel::execution::tag_invoke (*C++ function*), 1021
hpx::parallel::execution::then_execute (*C++ member*), 1004
hpx::parallel::execution::then_execute_t (*C++ struct*), 1008
hpx::parallel::execution::then_execute_t::tag_fallback_invoke (*C++ function*), 1009
hpx::parallel::execution::timed_executor (*C++ struct*), 1195
hpx::parallel::execution::timer_pool_executor (*C++ struct*), 1101
hpx::parallel::execution::timer_pool_executor::timer_pool_invoke (*C++ function*), 1101
hpx::parallel::execution::with_processing_units_count (*C++ function*), 878
hpx::parallel::execution::service_executor_type (*C++ member*), 993
hpx::parallel::execution::with_processing_units_count_t (*C++ struct*), 996
hpx::parallel::execution::with_processing_units_count_t (*C++ type*), 1407
hpx::parallel::util::concat (*C++ function*), 882
hpx::parallel::util::construct (*C++ function*), 875
hpx::parallel::util::construct_object (*C++ function*), 875
hpx::parallel::util::destroy (*C++ function*), 876
hpx::parallel::util::destroy_object (*C++ function*), 876
hpx::parallel::util::destroy_range (*C++ function*), 875
hpx::parallel::util::full_merge (*C++ function*), 875
hpx::parallel::util::full_merge4 (*C++ function*), 878
hpx::parallel::util::half_merge (*C++ function*), 877, 883
hpx::parallel::util::in_place_merge (*C++ function*), 877, 883
hpx::parallel::util::in_place_merge_uncontiguous (*C++ function*), 877, 883
hpx::parallel::util::init (*C++ function*), 875, 882
hpx::parallel::util::init_move (*C++ function*), 876, 882
hpx::parallel::util::is_mergeable (*C++ function*), 875
hpx::parallel::util::tag_invoke (*C++ function*), 879
hpx::parallel::util::less_range (*C++ function*), 878
hpx::parallel::util::merge_flow (*C++ function*), 879
hpx::parallel::util::merge_level4 (*C++ function*), 879
hpx::parallel::util::merge_vector4 (*C++ function*), 879
hpx::parallel::util::nbits32 (*C++ function*), 880
hpx::parallel::util::nbits64 (*C++ function*), 880
hpx::parallel::util::projection_identity (*C++ struct*), 881
hpx::parallel::util::projection_identity::is_transparent (*C++ function*), 881
hpx::parallel::util::tag_invoke (*C++ function*), 881
hpx::parallel::util::projection_identity::operator() (*C++ function*), 881
hpx::parallel::util::range (*C++ type*), 881
hpx::parallel::util::tmsb (*C++ member*), 880
hpx::parallel::util::uninit_full_merge (*C++ function*), 876, 883
hpx::parallel::util::uninit_full_merge4 (*C++ function*), 878

hpx::parallel::util::uninit_merge_level4
 (C++ function), 879

hpx::parallel::util::uninit_move (C++ function), 876, 882

hpx::parallel::v1 (C++ type), 417, 461, 1016, 1401–1409, 1412

hpx::parallel::v1::reduce_by_key (C++ function), 417

hpx::parallel::v1::sort_by_key (C++ function), 461

hpx::parallel::v2 (C++ type), 285, 352, 353

hpx::parallel::v2::task_block (C++ class), 285

hpx::parallel::v2::task_block::execution_policy
 (C++ type), 286

hpx::parallel::v2::task_block::get_execution_policy
 (C++ function), 286

hpx::parallel::v2::task_block::id_ (C++ member), 287

hpx::parallel::v2::task_block::policy (C++ function), 287

hpx::parallel::v2::task_block::policy_ (C++ member), 287

hpx::parallel::v2::task_block::run (C++ function), 286

hpx::parallel::v2::task_block::tasks_ (C++ member), 287

hpx::parallel::v2::task_block::wait (C++ function), 287

hpx::parallel::v2::task_canceled_exception
 (C++ class), 287

hpx::parallel::v2::task_canceled_exception::task_canceled_exception
 (C++ function), 288

hpx::parcelset (C++ type), 1337

hpx::parcelset::parcel_write_handler_type
 (C++ type), 1337

hpx::parcelset::parcelport_background_mode
 (C++ enum), 1338

hpx::parcelset::parcelport_background_mode::parcelport_background_mode::counter_path_elements
 (C++ enumerator), 1338

hpx::parcelset::parcelport_background_mode::parcelport_background_mode::fls_counter_path_elements::base_type
 (C++ enumerator), 1338

hpx::parcelset::parcelport_background_mode::parcelport_background_mode::receiver_path_elements::counter_path_elements
 (C++ enumerator), 1338

hpx::parcelset::parcelport_background_mode::parcelport_background_mode::sender_path_elements::instance
 (C++ enumerator), 1338

hpx::partial_sort (C++ function), 401

hpx::partial_sort_copy (C++ function), 402, 403

hpx::partition (C++ function), 404, 405

hpx::partition_copy (C++ function), 408, 409

hpx::performance_counters (C++ type), 1338, 1340, 1348, 1357, 1360

hpx::performance_counters::add_counter_type
 (C++ function), 1344, 1352

hpx::performance_counters::aqas_counter_discoverer::performance_counters::counter_path_elements::serializ

 (C++ function), 1340

hpx::performance_counters::agas_raw_counter_creator
 (C++ function), 1340

hpx::performance_counters::complement_counter_info
 (C++ function), 1352

hpx::performance_counters::counter_aggregating
 (C++ member), 1354

hpx::performance_counters::counter_average_base
 (C++ member), 1354

hpx::performance_counters::counter_average_count
 (C++ member), 1354

hpx::performance_counters::counter_average_timer
 (C++ member), 1354

hpx::performance_counters::counter_elapsed_time
 (C++ member), 1354

hpx::performance_counters::counter_histogram
 (C++ member), 1354

hpx::performance_counters::counter_info
 (C++ struct), 1344

hpx::performance_counters::counter_info::counter_info
 (C++ function), 1345

hpx::performance_counters::counter_info::fullname_

 (C++ member), 1345

hpx::performance_counters::counter_info::helptext_

 (C++ member), 1345

hpx::performance_counters::counter_info::serialize
 (C++ function), 1345

hpx::performance_counters::counter_info::status_

 (C++ member), 1345

hpx::performance_counters::counter_info::type_

 (C++ member), 1345

hpx::performance_counters::counter_info::unit_of_measure_

 (C++ member), 1345

hpx::performance_counters::counter_info::version_

 (C++ member), 1345

hpx::performance_counters::counter_monotonically_increasing

 (C++ member), 1345

hpx::parcelport_background_mode::counter_path_elements
 (C++ struct), 1345

hpx::parcelport_background_mode::fls_counter_path_elements::base_type
 (C++ type), 1346

hpx::parcelport_background_mode::receiver_path_elements::counter_path_elements
 (C++ function), 1346

hpx::parcelport_background_mode::sender_path_elements::instance
 (C++ member), 1346

hpx::performance_counters::counter_path_elements::instance
 (C++ member), 1346

hpx::performance_counters::counter_path_elements::parenting

 (C++ member), 1346

hpx::performance_counters::counter_path_elements::parenting

 (C++ member), 1346

hpx::performance_counters::counter_path_elements::parenting

 (C++ member), 1346

```

(C++ function), 1347
hpx::performance_counters::counter_path_element hpx::performance_counters::counter_type_path_elements::cou
(C++ member), 1346 (C++ function), 1347
hpx::performance_counters::counter_path_element hpx::performance_counters::counter_type_path_elements::cou
(C++ member), 1346 (C++ member), 1347
hpx::performance_counters::counter_prefix     hpx::performance_counters::counter_type_path_elements::obj
(C++ member), 1344 (C++ member), 1347
hpx::performance_counters::counter_prefix_len hpx::performance_counters::counter_type_path_elements::par
(C++ member), 1344 (C++ member), 1347
hpx::performance_counters::counter_raw (C++ member), 1344 hpx::performance_counters::counter_type_path_elements::ser
(C++ member), 1354 (C++ function), 1347
hpx::performance_counters::counter_raw_values hpx::performance_counters::counter_value
(C++ member), 1354 (C++ struct), 1354
hpx::performance_counters::counter_status     hpx::performance_counters::counter_value::count_
(C++ enum), 1343, 1350 (C++ member), 1355
hpx::performance_counters::counter_status::allready hpx::performance_counters::counter_value::counter_value
(C++ enumerator), 1343, 1344, 1350, 1351 (C++ function), 1355
hpx::performance_counters::counter_status::counter_type hpx::performance_counters::counter_value::get_value
(C++ enumerator), 1343, 1344, 1351 (C++ function), 1355
hpx::performance_counters::counter_status::counter_type::perf hpx::performance_counters::counter_value::scale_inverse_
(C++ enumerator), 1343, 1344, 1350, 1351 (C++ member), 1355
hpx::performance_counters::counter_status::genperf hpx::performance_counters::counter_value::scaling_
(C++ enumerator), 1343, 1344, 1351 (C++ member), 1355
hpx::performance_counters::counter_status::invalid hpx::performance_counters::counter_value::serialize
(C++ enumerator), 1343, 1344, 1350, 1351 (C++ function), 1355
hpx::performance_counters::counter_status::newperf hpx::performance_counters::counter_value::status_
(C++ enumerator), 1343, 1344, 1350, 1351 (C++ member), 1355
hpx::performance_counters::counter_status::valid hpx::performance_counters::counter_value::time_
(C++ enumerator), 1343, 1350, 1351 (C++ member), 1355
hpx::performance_counters::counter_text      hpx::performance_counters::counter_value::value_
(C++ member), 1354 (C++ member), 1355
hpx::performance_counters::counter_type      hpx::performance_counters::counter_values_array
(C++ enum), 1341, 1348 (C++ struct), 1355
hpx::performance_counters::counter_type::aggregat hpx::performance_counters::counter_values_array::count_
(C++ enumerator), 1342, 1343, 1349, 1350 (C++ member), 1356
hpx::performance_counters::counter_type::averag hpx::performance_counters::counter_values_array::counter_v
(C++ enumerator), 1341, 1343, 1348, 1350 (C++ function), 1356
hpx::performance_counters::counter_type::averag hpx::performance_counters::counter_values_array::get_value
(C++ enumerator), 1341, 1343, 1349, 1350 (C++ function), 1356
hpx::performance_counters::counter_type::averag hpx::performance_counters::counter_values_array::scale_inv
(C++ enumerator), 1342, 1343, 1349, 1350 (C++ member), 1356
hpx::performance_counters::counter_type::elaps hpx::performance_counters::counter_values_array::scaling_
(C++ enumerator), 1342, 1343, 1349, 1350 (C++ member), 1356
hpx::performance_counters::counter_type::histogr hpx::performance_counters::counter_values_array::serialize
(C++ enumerator), 1342, 1343, 1349, 1350 (C++ function), 1356
hpx::performance_counters::counter_type::monotoni hpx::performance_counters::counter_values_array::status_
(C++ enumerator), 1341, 1342, 1348, 1350 (C++ member), 1356
hpx::performance_counters::counter_type::raw   hpx::performance_counters::counter_values_array::time_
(C++ enumerator), 1341, 1342, 1348, 1350 (C++ member), 1356
hpx::performance_counters::counter_type::raw_val hpx::performance_counters::counter_values_array::values_
(C++ enumerator), 1342, 1343, 1350 (C++ member), 1356
hpx::performance_counters::counter_type::text   hpx::performance_counters::create_counter_func
(C++ enumerator), 1341, 1342, 1348, 1350 (C++ type), 1341
hpx::performance_counters::counter_type_path_ele hpx::performance_counters::default_counter_discoverer

```

(C++ function), 1339
hpx::performance_counters::discover_counter_type (C++ type), 1341
hpx::performance_counters::discover_counter_type (C++ function), 1352, 1353
hpx::performance_counters::discover_counter_type (C++ function), 1352
hpx::performance_counters::discover_counters (C++ type), 1341
hpx::performance_counters::discover_counters_mode (C++ enum), 1351
hpx::performance_counters::discover_counters_mode (C++ enumerator), 1351
hpx::performance_counters::discover_counters_mode (C++ enumerator), 1351
hpx::performance_counters::ensure_counter_prefix (C++ function), 1344
hpx::performance_counters::expand_counter_info (C++ function), 1353
hpx::performance_counters::get_counter (C++ function), 1344
hpx::performance_counters::get_counter_async (C++ function), 1353
hpx::performance_counters::get_counter_infos (C++ function), 1353
hpx::performance_counters::get_counter_instance (C++ function), 1352
hpx::performance_counters::get_counter_name (C++ function), 1351, 1352
hpx::performance_counters::get_counter_path_element (C++ function), 1352
hpx::performance_counters::get_counter_type (C++ function), 1353
hpx::performance_counters::get_counter_type_name (C++ function), 1344, 1351, 1352
hpx::performance_counters::get_counter_type_path (C++ function), 1352
hpx::performance_counters::get_full_counter_type (C++ function), 1351
hpx::performance_counters::install_counter_type (C++ function), 1357–1359
hpx::performance_counters::local_action_invocation (C++ function), 1340
hpx::performance_counters::local_action_invocation (C++ function), 1340
hpx::performance_counters::locality0_counter_discovery (C++ function), 1339
hpx::performance_counters::locality_counter_discovery (C++ function), 1339
hpx::performance_counters::locality_numa_counter_discovery (C++ function), 1339
hpx::performance_counters::locality_pool_counter_discovery (C++ function), 1339
hpx::performance_counters::locality_pool_thread_discovery (C++ function), 1339
hpx::performance_counters::locality_pool_thread_no_total_discovery (C++ function), 1339
hpx::performance_counters::locality_raw_counter_creator (C++ function), 1340
hpx::performance_counters::locality_raw_values_counter_creator (C++ function), 1340
hpx::performance_counters::locality_thread_counter_discovery (C++ function), 1339
hpx::performance_counters::operator> (C++ function), 1351
hpx::performance_counters::operator< (C++ function), 1351
hpx::performance_counters::operator<< (C++ function), 1351
hpx::performance_counters::registry (C++ class), 1360
hpx::performance_counters::registry::add_counter (C++ function), 1362
hpx::performance_counters::registry::add_counter_type (C++ function), 1360
hpx::performance_counters::registry::clear (C++ function), 1360
hpx::performance_counters::registry::counter_data (C++ struct), 1363
hpx::performance_counters::registry::counter_data::counter (C++ function), 1363
hpx::performance_counters::registry::counter_data::create (C++ member), 1363
hpx::performance_counters::registry::counter_data::discover (C++ member), 1363
hpx::performance_counters::registry::counter_data::info (C++ member), 1363
hpx::performance_counters::registry::counter_type_map_type (C++ type), 1363
hpx::performance_counters::registry::counter_types (C++ member), 1363
hpx::performance_counters::registry::create_arithmetics_counter (C++ function), 1362
hpx::performance_counters::registry::create_arithmetics_counter (C++ function), 1362
hpx::performance_counters::registry::create_counter (C++ function), 1361
hpx::performance_counters::registry::create_raw_counter (C++ function), 1361
hpx::performance_counters::registry::create_raw_counter_value (C++ function), 1361
hpx::performance_counters::registry::create_statistics_counter (C++ function), 1362
hpx::performance_counters::registry::discover_counter_type (C++ function), 1361
hpx::performance_counters::registry::discover_counter_type (C++ function), 1360
hpx::performance_counters::registry::get_counter_create_function (C++ function), 1360

(C++ function), 1361
`hpx::performance_counters::registry::get_counter` (C++ function), 1361
`hpx::performance_counters::registry::get_counter` (C++ function), 1362
`hpx::performance_counters::registry::instance` (C++ function), 1362
`hpx::performance_counters::registry::locate_counter` (C++ function), 1362
`hpx::performance_counters::registry::remove_counter` (C++ function), 1362
`hpx::performance_counters::registry::remove_counter` (C++ function), 1361
`hpx::performance_counters::remove_counter_prefix` (C++ function), 1344
`hpx::performance_counters::remove_counter_type` (C++ function), 1353
`hpx::performance_counters::status_already_defined` (C++ member), 1354
`hpx::performance_counters::status_counter_type` (C++ member), 1354
`hpx::performance_counters::status_counter_unknown` (C++ member), 1354
`hpx::performance_counters::status_generic_error` (C++ member), 1354
`hpx::performance_counters::status_invalid_data` (C++ member), 1354
`hpx::performance_counters::status_is_valid` (C++ function), 1344
`hpx::performance_counters::status_new_data` (C++ member), 1354
`hpx::performance_counters::status_valid_data` (C++ member), 1354
`hpx::PhonyNameDueToError::call` (C++ function), 1238
`hpx::placeholders` (C++ type), 1030
`hpx::placeholders::_1` (C++ member), 1030
`hpx::placeholders::_2` (C++ member), 1030
`hpx::placeholders::_3` (C++ member), 1030
`hpx::placeholders::_4` (C++ member), 1030
`hpx::placeholders::_5` (C++ member), 1030
`hpx::placeholders::_6` (C++ member), 1030
`hpx::placeholders::_7` (C++ member), 1030
`hpx::placeholders::_8` (C++ member), 1030
`hpx::placeholders::_9` (C++ member), 1030
`hpx::plain` (C++ member), 985
`hpx::plugins` (C++ type), 1079, 1363, 1365
`hpx::plugins::binary_filter_factory` (C++ struct), 1363
`hpx::plugins::binary_filter_factory::~binary_filter_factory` (C++ function), 1364
`hpx::plugins::binary_filter_factory::create` (C++ function), 1364
`hpx::plugins::binary_filter_factory::global_settings` (C++ member), 1364
`hpx::plugins::binary_filter_factory::isenabled` (C++ member), 1364
`hpx::plugins::binary_filter_factory::local_settings` (C++ member), 1364
`hpx::plugins::binary_filter_factory::plugin_registry` (C++ struct), 1365
`hpx::plugins::binary_filter_factory::get_plugin_info` (C++ function), 1365
`hpx::plugins::binary_filter_factory::plugin_registry_base` (C++ struct), 1079
`hpx::plugins::binary_filter_factory::~plugin_registry_base` (C++ function), 1079
`hpx::plugins::plugin_registry_base::get_plugin_info` (C++ function), 1079
`hpx::plugins::plugin_registry_base::init` (C++ function), 1079
`hpx::promise<T>` (C++ class), 1053
`hpx::promise::operator=` (C++ function), 1053
`hpx::promise::swap` (C++ function), 1053
`hpx::promise<R&>` (C++ class), 1054
`hpx::promise<R&>::~promise` (C++ function), 1054
`hpx::promise<R&>::base_type` (C++ type), 1054
`hpx::promise<R&>::operator=` (C++ function), 1054
`hpx::promise<R&>::promise` (C++ function), 1054
`hpx::promise<R&>::set_value` (C++ function), 1054
`hpx::promise<R&>::swap` (C++ function), 1054
`hpx::promise<void>` (C++ class), 1054
`hpx::promise<void>::~promise` (C++ function), 1054
`hpx::promise<void>::base_type` (C++ type), 1055
`hpx::promise<void>::operator=` (C++ function), 1054
`hpx::promise<void>::promise` (C++ function), 1054
`hpx::promise<void>::set_value` (C++ function), 1054
`hpx::promise<void>::swap` (C++ function), 1054
`hpx::ranges` (C++ type), 505, 510, 514, 524, 531, 538, 541, 545, 550, 555, 559, 579, 585, 595, 599, 603, 613, 619, 622, 630, 635, 641, 663, 669, 672, 676, 680, 685, 698, 708, 716, 733, 738, 744, 754, 761, 768, 775, 782, 785, 788, 792, 796, 800, 803, 814, 820, 845, 850, 853, 857, 862, 866
`hpx::ranges::adjacent_difference` (C++ func-

tion), 506–509
hpx::ranges::adjacent_find (*C++ function*), 511–513
hpx::ranges::all_of (*C++ function*), 520–523
hpx::ranges::any_of (*C++ function*), 517–520
hpx::ranges::copy (*C++ function*), 524–526
hpx::ranges::copy_if (*C++ function*), 527, 529, 530
hpx::ranges::copy_n (*C++ function*), 526, 527
hpx::ranges::count (*C++ function*), 531–534
hpx::ranges::count_if (*C++ function*), 534, 535, 537
hpx::ranges::destroy (*C++ function*), 538–540
hpx::ranges::destroy_n (*C++ function*), 540, 541
hpx::ranges::ends_with (*C++ function*), 542, 544
hpx::ranges::equal (*C++ function*), 546–549
hpx::ranges::exclusive_scan (*C++ function*), 550, 551, 553
hpx::ranges::experimental (*C++ type*), 585
hpx::ranges::experimental::for_loop (*C++ function*), 585–588
hpx::ranges::experimental::for_loop_strided (*C++ function*), 589, 591–593
hpx::ranges::fill (*C++ function*), 555–557
hpx::ranges::fill_n (*C++ function*), 557–559
hpx::ranges::find (*C++ function*), 559–561
hpx::ranges::find_end (*C++ function*), 569, 570, 572, 573
hpx::ranges::find_first_of (*C++ function*), 574, 575, 577, 578
hpx::ranges::find_if (*C++ function*), 562–565
hpx::ranges::find_if_not (*C++ function*), 565–568
hpx::ranges::for_each (*C++ function*), 579–581
hpx::ranges::for_each_n (*C++ function*), 583
hpx::ranges::generate (*C++ function*), 595–597
hpx::ranges::generate_n (*C++ function*), 597, 598
hpx::ranges::includes (*C++ function*), 599–602
hpx::ranges::inclusive_scan (*C++ function*), 604, 606–611
hpx::ranges::inplace_merge (*C++ function*), 647–650
hpx::ranges::is_heap (*C++ function*), 613–615
hpx::ranges::is_heap_until (*C++ function*), 616–618
hpx::ranges::is_partitioned (*C++ function*), 619–621
hpx::ranges::is_sorted (*C++ function*), 622–625
hpx::ranges::is_sorted_until (*C++ function*), 626, 628
hpx::ranges::lexicographical_compare (*C++ function*), 630–633
hpx::ranges::make_heap (*C++ function*), 635–640
hpx::ranges::merge (*C++ function*), 641, 642, 644, 646
hpx::ranges::mismatch (*C++ function*), 663, 665, 667, 668
hpx::ranges::move (*C++ function*), 669–671
hpx::ranges::none_of (*C++ function*), 514–517
hpx::ranges::nth_element (*C++ function*), 672–674
hpx::ranges::partial_sort (*C++ function*), 676–678
hpx::ranges::partial_sort_copy (*C++ function*), 680–682, 684
hpx::ranges::partition (*C++ function*), 685–688
hpx::ranges::partition_copy (*C++ function*), 693–695, 697
hpx::ranges::reduce (*C++ function*), 698–707
hpx::ranges::remove (*C++ function*), 712–714
hpx::ranges::remove_if (*C++ function*), 708–711
hpx::ranges::replace (*C++ function*), 720–722
hpx::ranges::replace_copy (*C++ function*), 728–731
hpx::ranges::replace_copy_if (*C++ function*), 723–725, 727
hpx::ranges::replace_if (*C++ function*), 716–719
hpx::ranges::reverse (*C++ function*), 733, 734
hpx::ranges::reverse_copy (*C++ function*), 735–737
hpx::ranges::rotate (*C++ function*), 738–740
hpx::ranges::rotate_copy (*C++ function*), 741–743
hpx::ranges::search (*C++ function*), 744, 745, 747, 748
hpx::ranges::search_n (*C++ function*), 749, 750, 752, 753
hpx::ranges::set_difference (*C++ function*), 754, 756, 758, 759
hpx::ranges::set_intersection (*C++ function*), 761, 763, 765, 766
hpx::ranges::set_symmetric_difference (*C++ function*), 768, 770, 772, 774
hpx::ranges::set_union (*C++ function*), 775, 777, 780
hpx::ranges::shift_left (*C++ function*), 782–784
hpx::ranges::shift_right (*C++ function*), 785–787
hpx::ranges::sort (*C++ function*), 788–790
hpx::ranges::stable_partition (*C++ function*), 689–692
hpx::ranges::stable_sort (*C++ function*), 792–794
hpx::ranges::starts_with (*C++ function*), 796–798
hpx::ranges::swap_ranges (*C++ function*), 800–802
hpx::ranges::tagFallback_invoke (*C++ function*), 779
hpx::ranges::transform (*C++ function*), 803–805, 809–812
hpx::ranges::transform_exclusive_scan (*C++ function*), 814, 815, 817, 818
hpx::ranges::transform_inclusive_scan (*C++ function*), 820, 821, 823–825, 827, 828, 830

hpx::ranges::transform_t (*C++ function*), 807
 hpx::ranges::uninitialized_copy (*C++ function*), 845–847
 hpx::ranges::uninitialized_copy_n (*C++ function*), 848
 hpx::ranges::uninitialized_default_construct (*C++ function*), 850, 851
 hpx::ranges::uninitialized_default_construct_r (*C++ function*), 852, 853
 hpx::ranges::uninitialized_fill (*C++ function*), 854, 855
 hpx::ranges::uninitialized_fill_n (*C++ function*), 856
 hpx::ranges::uninitialized_move (*C++ function*), 858–860
 hpx::ranges::uninitialized_move_n (*C++ function*), 861
 hpx::ranges::uninitialized_value_construct (*C++ function*), 862–864
 hpx::ranges::uninitialized_value_construct_n (*C++ function*), 865
 hpx::ranges::unique (*C++ function*), 866–869
 hpx::ranges::unique_copy (*C++ function*), 870–873
 hpx::recursive_mutex (*C++ type*), 1145
 hpx::reduce (*C++ function*), 411–413, 415, 416
 hpx::register_on_exit (*C++ function*), 1099
 hpx::register_pre_shutdown_function (*C++ function*), 1101
 hpx::register_pre_startup_function (*C++ function*), 1103
 hpx::register_shutdown_function (*C++ function*), 1102
 hpx::register_startup_function (*C++ function*), 1103
 hpx::register_thread (*C++ function*), 1098
 hpx::register_thread_on_error_func (*C++ function*), 1105
 hpx::register_thread_on_start_func (*C++ function*), 1104
 hpx::register_thread_on_stop_func (*C++ function*), 1105
 hpx::register_with_basename (*C++ function*), 1298, 1299
 hpx::remove (*C++ function*), 420
 hpx::remove_copy (*C++ function*), 423
 hpx::remove_copy_if (*C++ function*), 424, 425
 hpx::remove_if (*C++ function*), 421, 422
 hpx::replace (*C++ function*), 427
 hpx::replace_copy (*C++ function*), 430, 431
 hpx::replace_copy_if (*C++ function*), 432
 hpx::replace_if (*C++ function*), 428, 429
 hpx::report_error (*C++ function*), 1090
 hpx::resiliency (*C++ type*), 1072, 1074
 hpx::resiliency::experimental (*C++ type*), 1072, 1074
 hpx::resiliency::experimental::make_replay_executor (*C++ function*), 1072
 hpx::resiliency::experimental::make_replicate_executor (*C++ function*), 1075
 hpx::resiliency::experimental::replay_executor (*C++ class*), 1073
 hpx::resiliency::experimental::replay_executor::context (*C++ function*), 1073
 hpx::resiliency::experimental::replay_executor::exec_ (*C++ member*), 1074
 hpx::resiliency::experimental::replay_executor::execution_ (*C++ type*), 1073
 hpx::resiliency::experimental::replay_executor::executor_p_ (*C++ type*), 1073
 hpx::resiliency::experimental::replay_executor::future_type_ (*C++ type*), 1073
 hpx::resiliency::experimental::replay_executor::num_spread_ (*C++ member*), 1073
 hpx::resiliency::experimental::replay_executor::num_tasks_ (*C++ member*), 1073
 hpx::resiliency::experimental::replay_executor::operator!= (*C++ function*), 1073
 hpx::resiliency::experimental::replay_executor::operator== (*C++ function*), 1073
 hpx::resiliency::experimental::replay_executor::replay_count_ (*C++ member*), 1074
 hpx::resiliency::experimental::replay_executor::replay_exec_ (*C++ function*), 1073
 hpx::resiliency::experimental::replay_executor::tag_invoke_ (*C++ function*), 1073
 hpx::resiliency::experimental::replay_executor::validator_ (*C++ member*), 1074
 hpx::resiliency::experimental::replicate_executor (*C++ class*), 1075
 hpx::resiliency::experimental::replicate_executor::context_ (*C++ function*), 1076
 hpx::resiliency::experimental::replicate_executor::exec_ (*C++ member*), 1076
 hpx::resiliency::experimental::replicate_executor::executing_ (*C++ type*), 1075
 hpx::resiliency::experimental::replicate_executor::execute_ (*C++ type*), 1075
 hpx::resiliency::experimental::replicate_executor::future_type_ (*C++ type*), 1075
 hpx::resiliency::experimental::replicate_executor::num_spread_ (*C++ member*), 1076
 hpx::resiliency::experimental::replicate_executor::num_tasks_ (*C++ member*), 1076
 hpx::resiliency::experimental::replicate_executor::operator!= (*C++ function*), 1076
 hpx::resiliency::experimental::replicate_executor::operator== (*C++ function*), 1076
 hpx::resiliency::experimental::replicate_executor::operator_ (*C++ function*), 1076
 hpx::resiliency::experimental::replicate_executor::replica_

(C++ member), 1076
hpx::resiliency::experimental::replicate_executor (C++ function),
 (C++ function), 1076
hpx::resiliency::experimental::replicate_executor (C++ function),
 (C++ function), 1076
hpx::resiliency::experimental::replicate_executor (C++ function),
 (C++ member), 1076
hpx::resiliency::experimental::replicate_executor (C++ function),
 (C++ member), 1076
hpx::resource (C++ type), 1105
hpx::resource::get_num_thread_pools (C++ function),
 (C++ function), 1106
hpx::resource::get_num_threads (C++ function),
 1106
hpx::resource::get_pool_index (C++ function),
 1106
hpx::resource::get_pool_name (C++ function),
 1106
hpx::resource::get_thread_pool (C++ function),
 1106
hpx::resource::pool_exists (C++ function), 1106
hpx::resume (C++ function), 1336
hpx::rethrow (C++ member), 985
hpx::reverse (C++ function), 434
hpx::reverse_copy (C++ function), 435, 436
hpx::rotate (C++ function), 437
hpx::rotate_copy (C++ function), 438, 439
hpx::runtime (C++ class), 1090
hpx::runtime::~runtime (C++ function), 1091
hpx::runtime::add_pre_shutdown_function
 (C++ function), 1093
hpx::runtime::add_pre_startup_function (C++
 function), 1093
hpx::runtime::add_shutdown_function (C++
 function), 1093
hpx::runtime::add_startup_function (C++
 function), 1093
hpx::runtime::assign_cores (C++ function), 1095
hpx::runtime::call_startup_functions (C++
 function), 1097
hpx::runtime::deinit_global_data (C++
 function), 1095
hpx::runtime::deinit_tss_helper (C++ function),
 1097
hpx::runtime::enumerate_os_threads (C++
 function), 1094
hpx::runtime::exception_ (C++ member), 1096
hpx::runtime::finalize (C++ function), 1092
hpx::runtime::get_config (C++ function), 1091
hpx::runtime::get_initial_num_localities
 (C++ function), 1095
hpx::runtime::get_instance_number (C++
 function), 1091
hpx::runtime::get_locality_id (C++ function),
 1095
hpx::runtime::replicate_executor_name (C++ function),
 1095
hpx::runtime::replicate_executor_notify (C++
 function), 1091
hpx::runtime::replicate_executor_notify_num_localities (C++
 function), 1095
hpx::runtime::runtime::get_num_worker_threads (C++
 function), 1095
hpx::runtime::get_os_thread_data (C++
 function), 1094
hpx::runtime::get_state (C++ function), 1091
hpx::runtime::get_system_uptime (C++ function),
 1095
hpx::runtime::get_thread_manager (C++
 function), 1092
hpx::runtime::get_thread_mapper (C++ function),
 1091
hpx::runtime::get_thread_pool (C++ function),
 1094
hpx::runtime::get_topology (C++ function), 1091
hpx::runtime::here (C++ function), 1092
hpx::runtime::hpx_errorsink_function_type
 (C++ type), 1090
hpx::runtime::hpx_main_function_type (C++
 type), 1090
hpx::runtime::init (C++ function), 1095
hpx::runtime::init_global_data (C++ function),
 1095
hpx::runtime::init_tss_ex (C++ function), 1097
hpx::runtime::init_tss_helper (C++ function),
 1097
hpx::runtime::instance_number_ (C++ member),
 1096
hpx::runtime::instance_number_counter_ (C++
 member), 1096
hpx::runtime::is_networking_enabled (C++
 function), 1092
hpx::runtime::main_pool_ (C++ member), 1096
hpx::runtime::main_pool_notifier_ (C++ mem-
 ber), 1096
hpx::runtime::mtx_ (C++ member), 1096
hpx::runtime::notification_policy_type (C++
 type), 1090
hpx::runtime::notifier_ (C++ member), 1096
hpx::runtime::notify_finalize (C++ function),
 1097
hpx::runtime::on_error_func (C++ function),
 1094, 1095
hpx::runtime::on_error_func_ (C++ member),
 1096
hpx::runtime::on_exit (C++ function), 1091
hpx::runtime::on_exit_functions_ (C++ mem-
 ber), 1096

hpx::runtime::on_exit_type (C++ type), 1095
hpx::runtime::on_start_func (C++ function), 1094, 1095
hpx::runtime::on_start_func_ (C++ member), 1096
hpx::runtime::on_stop_func (C++ function), 1094, 1095
hpx::runtime::on_stop_func_ (C++ member), 1096
hpx::runtime::pre_shutdown_functions_ (C++ member), 1097
hpx::runtime::pre_startup_functions_ (C++ member), 1097
hpx::runtime::register_thread (C++ function), 1094
hpx::runtime::report_error (C++ function), 1093
hpx::runtime::result_ (C++ member), 1096
hpx::runtime::resume (C++ function), 1092
hpx::runtime::rethrow_exception (C++ function), 1092
hpx::runtime::rtcfg_ (C++ member), 1096
hpx::runtime::run (C++ function), 1091
hpx::runtime::run_helper (C++ function), 1095
hpx::runtime::runtime (C++ function), 1091, 1095
hpx::runtime::set_notification_policies (C++ function), 1095
hpx::runtime::set_state (C++ function), 1091
hpx::runtime::shutdown_functions_ (C++ member), 1097
hpx::runtime::start (C++ function), 1092
hpx::runtime::starting (C++ function), 1091
hpx::runtime::startup_functions_ (C++ member), 1097
hpx::runtime::state_ (C++ member), 1096
hpx::runtime::stop (C++ function), 1092
hpx::runtime::stop_called_ (C++ member), 1097
hpx::runtime::stop_done_ (C++ member), 1097
hpx::runtime::stop_helper (C++ function), 1097
hpx::runtime::stopped (C++ function), 1091
hpx::runtime::stopping (C++ function), 1091
hpx::runtime::suspend (C++ function), 1092
hpx::runtime::thread_manager_ (C++ member), 1096
hpx::runtime::thread_support_ (C++ member), 1096
hpx::runtime::topology_ (C++ member), 1096
hpx::runtime::unregister_thread (C++ function), 1094
hpx::runtime::wait (C++ function), 1092
hpx::runtime::wait_condition_ (C++ member), 1097
hpx::runtime::wait_finalize (C++ function), 1097
hpx::runtime::wait_helper (C++ function), 1095
hpx::runtime_distributed (C++ class), 1374
hpx::runtime_distributed::~runtime_distributed (C++ function), 1374
hpx::runtime_distributed::active_counters_ (C++ member), 1379
hpx::runtime_distributed::add_pre_shutdown_function (C++ function), 1377
hpx::runtime_distributed::add_pre_startup_function (C++ function), 1377
hpx::runtime_distributed::add_shutdown_function (C++ function), 1377
hpx::runtime_distributed::add_startup_function (C++ function), 1377
hpx::runtime_distributed::agas_client_ (C++ member), 1379
hpx::runtime_distributed::applier_ (C++ member), 1379
hpx::runtime_distributed::assign_cores (C++ function), 1378
hpx::runtime_distributed::default_errorsink (C++ function), 1379
hpx::runtime_distributed::deinit_global_data (C++ function), 1378
hpx::runtime_distributed::deinit_tss_helper (C++ function), 1379
hpx::runtime_distributed::evaluate_active_counters (C++ function), 1376
hpx::runtime_distributed::finalize (C++ function), 1375
hpx::runtime_distributed::get_agas_client (C++ function), 1377
hpx::runtime_distributed::get_applier (C++ function), 1377
hpx::runtime_distributed::get_counter_registry (C++ function), 1376
hpx::runtime_distributed::get_id_pool (C++ function), 1377
hpx::runtime_distributed::get_initial_num_localities (C++ function), 1378
hpx::runtime_distributed::get_locality_id (C++ function), 1378
hpx::runtime_distributed::get_locality_name (C++ function), 1378
hpx::runtime_distributed::get_next_id (C++ function), 1377
hpx::runtime_distributed::get_notification_policy (C++ function), 1378
hpx::runtime_distributed::get_num_localities (C++ function), 1378
hpx::runtime_distributed::get_num_worker_threads (C++ function), 1378
hpx::runtime_distributed::get_runtime_support_lva (C++ function), 1377
hpx::runtime_distributed::get_thread_manager (C++ function), 1377

hpx::runtime_distributed::get_thread_pool
 (C++ function), 1378

hpx::runtime_distributed::here (C++ function),
 1377

hpx::runtime_distributed::id_pool_ (C++ mem-
ber), 1379

hpx::runtime_distributed::init_global_data
 (C++ function), 1378

hpx::runtime_distributed::init_id_pool_range
 (C++ function), 1377

hpx::runtime_distributed::init_tss_ex (C++
function), 1379

hpx::runtime_distributed::init_tss_helper
 (C++ function), 1378

hpx::runtime_distributed::initialize_agas
 (C++ function), 1377

hpx::runtime_distributed::is_networking_enabled
 (C++ function), 1376

hpx::runtime_distributed::mode_ (C++ member),
 1379

hpx::runtime_distributed::post_main_ (C++
member), 1379

hpx::runtime_distributed::pre_main_ (C++
member), 1379

hpx::runtime_distributed::register_counter_types
 (C++ function), 1376

hpx::runtime_distributed::register_query_counters
 (C++ function), 1376

hpx::runtime_distributed::register_thread
 (C++ function), 1378

hpx::runtime_distributed::reinit_active_counters
 (C++ function), 1376

hpx::runtime_distributed::report_error (C++
function), 1375

hpx::runtime_distributed::reset_active_counters
 (C++ function), 1376

hpx::runtime_distributed::resume (C++ func-
tion), 1375

hpx::runtime_distributed::run (C++ function),
 1376

hpx::runtime_distributed::run_helper (C++
function), 1378

hpx::runtime_distributed::runtime_distributed
 (C++ function), 1374

hpx::runtime_distributed::runtime_support_
 (C++ member), 1379

hpx::runtime_distributed::set_error_sink
 (C++ function), 1376

hpx::runtime_distributed::start (C++ function),
 1374, 1375

hpx::runtime_distributed::start_active_counters
 (C++ function), 1376

hpx::runtime_distributed::stop (C++ function),
 1375

hpx::runtime_distributed::stop_active_counters
 (C++ function), 1376

hpx::runtime_distributed::stop_evaluating_counters
 (C++ function), 1377

hpx::runtime_distributed::stop_helper (C++
function), 1375

hpx::runtime_distributed::suspend (C++ func-
tion), 1375

hpx::runtime_distributed::used_cores_map_
 (C++ member), 1379

hpx::runtime_distributed::used_cores_map_type
 (C++ type), 1378

hpx::runtime_distributed::wait (C++ function),
 1375

hpx::runtime_distributed::wait_helper (C++
function), 1378

hpx::runtime_mode (C++ enum), 1079

hpx::runtime_mode::connect (C++ enumerator),
 1080

hpx::runtime_mode::console (C++ enumerator),
 1079

hpx::runtime_mode::default_ (C++ enumerator),
 1080

hpx::runtime_mode::invalid (C++ enumerator),
 1079

hpx::runtime_mode::last (C++ enumerator), 1080

hpx::runtime_mode::local (C++ enumerator), 1080

hpx::runtime_mode::worker (C++ enumerator),
 1080

hpx::scoped_annotation (C++ struct), 1168

hpx::scoped_annotation::~scoped_annotation
 (C++ function), 1168

hpx::scoped_annotation::HPX_NON_COPYABLE
 (C++ function), 1168

hpx::scoped_annotation::scoped_annotation
 (C++ function), 1168

hpx::search (C++ function), 440, 441

hpx::search_n (C++ function), 442, 443

hpx::segmented (C++ type), 1401–1409, 1411, 1412

hpx::segmented::minmax_element_result (C++
type), 1407

hpx::segmented::tag_invoke (C++ function), 1402–
 1413

hpx::serialization (C++ type), 1030, 1032, 1050,
 1108, 1263

hpx::serialization::base_object (C++ function),
 1108

hpx::serialization::base_object_type (C++
struct), 1108

hpx::serialization::base_object_type::base_object_type
 (C++ function), 1108

hpx::serialization::base_object_type::d_
 (C++ member), 1108

hpx::serialization::base_object_type::serialize

(*C++ function*), 1108
 hpx::serialization::base_object_type<Derived, hpx::shared_future::base_type (*C++ type*), 1048
 Base, std::true_type> (*C++ struct*), 1107 hpx::shared_future::get (*C++ function*), 1048
 hpx::serialization::base_object_type<Derived, hpx::shared_future::operator= (*C++ function*),
 Base, std::true_type>::base_object_type 1048
 (*C++ function*), 1107 hpx::shared_future::result_type (*C++ type*),
 hpx::serialization::base_object_type<Derived, 1047
 Base, std::true_type>::d_ (*C++ member*), 1108 hpx::shared_future::shared_future (*C++ function*), 1048
 hpx::serialization::base_object_type<Derived, hpx::shared_future::shared_state_type (*C++ function*), 1047
 Base, std::true_type>::HPX_SERIALIZATION_SPLIT_MEMBER 1047
 (*C++ function*), 1107 hpx::shared_future::then (*C++ function*), 1048
 hpx::serialization::base_object_type<Derived, hpx::shared_future::then_alloc (*C++ function*),
 Base, std::true_type>::load (*C++ function*), 1107 1048
 hpx::shared_mutex (*C++ type*), 1145
 hpx::serialization::base_object_type<Derived, hpx::shift_left (*C++ function*), 455, 456
 Base, std::true_type>::save (*C++ function*), 1107 hpx::shift_right (*C++ function*), 457, 458
 hpx::serialization::operator& (*C++ function*), 1108 hpx::shutdown_function_type (*C++ type*), 1101
 hpx::serialization::operator>> (*C++ function*), 1108 hpx::sliding_semaphore (*C++ type*), 1146
 hpx::serialization::operator<< (*C++ function*), 1108 hpx::sliding_semaphore_var (*C++ class*), 1146
 hpx::serialization::PhonyNameDueToError::base_lco_sliding_semaphore_var::sem_ (*C++ member*), 1147 hpx::sliding_semaphore_var::mtx_ (*C++ member*), 1147
 hpx::serialization::PhonyNameDueToError::d_ (*C++ member*), 1109 hpx::sliding_semaphore_var::set_max_difference
 (*C++ function*), 1147
 hpx::serialization::PhonyNameDueToError::HPX_SERIALIZATION_SPLIT_MEMBER::signal (*C++ function*), 1147
 (*C++ function*), 1109
 hpx::serialization::PhonyNameDueToError::load hpx::sliding_semaphore_var::signal_all (*C++ function*), 1147
 (*C++ function*), 1109
 hpx::serialization::PhonyNameDueToError::save hpx::sliding_semaphore_var::sliding_semaphore_var
 (*C++ function*), 1109 (*C++ function*), 1147
 hpx::serialization::serialize (*C++ function*), 1031, 1032, 1050 hpx::sliding_semaphore_var::try_wait (*C++ function*), 1147
 hpx::set_custom_exception_info_handler (*C++ function*), 980 hpx::sliding_semaphore_var::wait (*C++ function*), 1147
 hpx::set_difference (*C++ function*), 444, 445 hpx::sort (*C++ function*), 459
 hpx::set_error_handlers (*C++ function*), 1090 hpx::source_location (*C++ struct*), 886
 hpx::set_intersection (*C++ function*), 447, 448 hpx::source_location::column (*C++ function*), 886
 hpx::set_lco_error (*C++ function*), 1253, 1254 hpx::source_location::file_name (*C++ function*),
 886
 hpx::set_lco_value (*C++ function*), 1251, 1252 hpx::source_location::filename (*C++ member*),
 887
 hpx::set_lco_value_unmanaged (*C++ function*), 1251, 1252 hpx::source_location::function_name (*C++ function*),
 886
 hpx::set_pre_exception_handler (*C++ function*), 980 hpx::source_location::functionname (*C++ member*),
 887
 hpx::set_symmetric_difference (*C++ function*), 450, 451 hpx::source_location::line (*C++ function*), 886
 hpx::set_thread_termination_handler (*C++ function*), 1160 hpx::source_location::line_number (*C++ member*),
 887
 hpx::set_union (*C++ function*), 453, 454 hpx::spinlock (*C++ type*), 1148
 hpx::shared_future (*C++ class*), 1047, 1050 hpx::spinlock_no_backoff (*C++ type*), 1148
 hpx::shared_future::~shared_future (*C++ function*)

hpx::split_future (*C++ function*), 892
hpx::stable_partition (*C++ function*), 406, 407
hpx::stable_sort (*C++ function*), 462, 463
hpx::start (*C++ function*), 1332–1334
hpx::starts_with (*C++ function*), 464, 465
hpx::startup_function_type (*C++ type*), 1102
hpx::stop (*C++ function*), 1327
hpx::stop_callback (*C++ class*), 1149
hpx::stop_callback (*C++ function*), 1148
hpx::stop_source (*C++ class*), 1149
hpx::stop_source::~stop_source (*C++ function*), 1149
hpx::stop_source::get_token (*C++ function*), 1150
hpx::stop_source::operator!= (*C++ function*), 1150
hpx::stop_source::operator= (*C++ function*), 1149
hpx::stop_source::operator== (*C++ function*), 1150
hpx::stop_source::request_stop (*C++ function*), 1150
hpx::stop_source::state_ (*C++ member*), 1150
hpx::stop_source::stop_possible (*C++ function*), 1150
hpx::stop_source::stop_requested (*C++ function*), 1150
hpx::stop_source::stop_source (*C++ function*), 1149
hpx::stop_source::swap (*C++ function*), 1149
hpx::stop_token (*C++ class*), 1150
hpx::stop_token::~stop_token (*C++ function*), 1151
hpx::stop_token::callback_type (*C++ type*), 1150
hpx::stop_token::operator!= (*C++ function*), 1151
hpx::stop_token::operator= (*C++ function*), 1151
hpx::stop_token::operator== (*C++ function*), 1151
hpx::stop_token::state_ (*C++ member*), 1151
hpx::stop_token::stop_possible (*C++ function*), 1151
hpx::stop_token::stop_requested (*C++ function*), 1151
hpx::stop_token::stop_token (*C++ function*), 1151
hpx::stop_token::swap (*C++ function*), 1151
hpx::suspend (*C++ function*), 1335
hpx::swap (*C++ function*), 1053, 1148, 1158, 1160
hpx::swap_ranges (*C++ function*), 467
hpx::sync (*C++ function*), 891
hpx::terminate (*C++ function*), 1326
hpx::this_thread (*C++ type*), 1015, 1163, 1176
hpx::this_thread::disable_interruption (*C++ class*), 1164
hpx::this_thread::disable_interruption::~disable_interruption (*C++ function*), 1165
hpx::this_thread::disable_interruption::operator!= (*C++ function*), 1165
hpx::this_thread::enable_interruption (*C++ class*), 1163
hpx::this_thread::enable_interruption::operator!= (*C++ function*), 1163
hpx::this_thread::enable_interruption::operator== (*C++ function*), 1163
hpx::this_thread::get_executor (*C++ function*), 1015
hpx::this_thread::get_id (*C++ function*), 1164
hpx::this_thread::get_pool (*C++ function*), 1178
hpx::this_thread::get_priority (*C++ function*), 1164
hpx::this_thread::get_stack_size (*C++ function*), 1164
hpx::this_thread::get_thread_data (*C++ function*), 1164
hpx::this_thread::interrupt (*C++ function*), 1164
hpx::this_thread::interruption_enabled (*C++ function*), 1164
hpx::this_thread::interruption_point (*C++ function*), 1164
hpx::this_thread::interruption_requested (*C++ function*), 1164
hpx::this_thread::restore_interruption (*C++ class*), 1165
hpx::this_thread::restore_interruption::~restore_interruption (*C++ function*), 1165
hpx::this_thread::restore_interruption::interruption_was_enabled (*C++ member*), 1165
hpx::this_thread::restore_interruption::operator!= (*C++ function*), 1165
hpx::this_thread::restore_interruption::operator= (*C++ function*), 1165
hpx::this_thread::restore_interruption::restore_interruption (*C++ function*), 1165
hpx::this_thread::set_thread_data (*C++ function*), 1164
hpx::this_thread::sleep_for (*C++ function*), 1164
hpx::this_thread::sleep_until (*C++ function*), 1164
hpx::this_thread::suspend (*C++ function*), 1176–1178
hpx::this_thread::yield (*C++ function*), 1164
hpx::this_thread::yield_to (*C++ function*), 1164
hpx::thread (*C++ class*), 1160
hpx::thread::~thread (*C++ function*), 1161
hpx::thread::detach (*C++ function*), 1161
hpx::thread::detach_locked (*C++ function*), 1162
hpx::thread::get_future (*C++ function*), 1162
hpx::thread::get_id (*C++ function*), 1161
hpx::thread::get_thread_data (*C++ function*), 1162
hpx::thread::hardware_concurrency (*C++ function*), 1162
hpx::thread::id::id (*C++ function*), 1163
hpx::thread::id::id_ (*C++ member*), 1163
hpx::thread::native_handle (*C++ function*),

1163
 hpx::thread::id::operator!=(*C++ function*), 1163
 hpx::thread::id::operator==(*C++ function*), 1163
 hpx::thread::id::operator>(*C++ function*), 1163
 hpx::thread::id::operator>=(*C++ function*), 1163
 hpx::thread::id::operator<(*C++ function*), 1163
 hpx::thread::id::operator<=(*C++ function*), 1163
 hpx::thread::id::operator<<(*C++ function*), 1163
 hpx::thread::id_ (*C++ member*), 1162
 hpx::thread::interrupt (*C++ function*), 1162
 hpx::thread::interruption_requested (*C++ function*), 1162
 hpx::thread::join (*C++ function*), 1161
 hpx::thread::joinable (*C++ function*), 1161
 hpx::thread::joinable_locked (*C++ function*), 1162
 hpx::thread::mtx_ (*C++ member*), 1162
 hpx::thread::mutex_type (*C++ type*), 1162
 hpx::thread::native_handle (*C++ function*), 1162
 hpx::thread::native_handle_type (*C++ type*), 1161
 hpx::thread::operator= (*C++ function*), 1161
 hpx::thread::set_thread_data (*C++ function*), 1162
 hpx::thread::start_thread (*C++ function*), 1162
 hpx::thread::swap (*C++ function*), 1161
 hpx::thread::terminate (*C++ function*), 1162
 hpx::thread::thread (*C++ function*), 1161
 hpx::thread::thread_function_nullary (*C++ function*), 1163
 hpx::thread_interrupted (*C++ struct*), 983
 hpx::thread_termination_handler_type (*C++ type*), 1160
 hpx::threads (*C++ type*), 944, 949, 1015, 1089, 1097, 1100, 1106, 1141, 1155, 1167, 1169, 1174, 1178, 1185, 1187, 1198
 hpx::threads::create_topology (*C++ function*), 1199
 hpx::threads::enumerate_threads (*C++ function*), 1107
 hpx::threads::get_ctx_ptr (*C++ function*), 1169
 hpx::threads::get_default_stack_size (*C++ function*), 1098
 hpx::threads::get_executor (*C++ function*), 1015
 hpx::threads::get_idle_core_count (*C++ function*), 1107
 hpx::threads::get_idle_core_mask (*C++ function*), 1107
 hpx::threads::get_memory_page_size (*C++ function*), 1199
 hpx::threads::get_parent_id (*C++ function*), 1169
 hpx::threads::get_parent_locality_id (*C++ function*), 1170
 hpx::threads::get_parent_phase (*C++ function*), 1169
 hpx::threads::get_pool (*C++ function*), 1183
 hpx::threads::get_self (*C++ function*), 1169
 hpx::threads::get_self_component_id (*C++ function*), 1170
 hpx::threads::get_self_id (*C++ function*), 1142, 1169
 hpx::threads::get_self_id_data (*C++ function*), 1169
 hpx::threads::get_self_ptr (*C++ function*), 1142, 1169
 hpx::threads::get_self_ptr_checked (*C++ function*), 1169
 hpx::threads::get_self_stacksize (*C++ function*), 1170
 hpx::threads::get_self_stacksize_enum (*C++ function*), 1170
 hpx::threads::get_stack_size (*C++ function*), 1098, 1183
 hpx::threads::get_stack_size_enum_name (*C++ function*), 948
 hpx::threads::get_stack_size_name (*C++ function*), 1098
 hpx::threads::get_thread_count (*C++ function*), 1106
 hpx::threads::get_thread_description (*C++ function*), 1174
 hpx::threads::get_thread_id_data (*C++ function*), 1169
 hpx::threads::get_thread_interruption_enabled (*C++ function*), 1181
 hpx::threads::get_thread_interruption_requested (*C++ function*), 1182
 hpx::threads::get_thread_lco_description (*C++ function*), 1175
 hpx::threads::get_thread_phase (*C++ function*), 1181
 hpx::threads::get_thread_priority (*C++ function*), 1182
 hpx::threads::get_thread_priority_name (*C++ function*), 947
 hpx::threads::get_thread_state (*C++ function*), 1180
 hpx::threads::get_thread_state_ex_name (*C++ function*), 948
 hpx::threads::get_thread_state_name (*C++ function*), 947, 948
 hpx::threads::hpx_hwloc_bitmap_wrapper (*C++ struct*), 1199
 hpx::threads::hpx_hwloc_bitmap_wrapper::~hpx_hwloc_bitmap_ (*C++ function*), 1199
 hpx::threads::hpx_hwloc_bitmap_wrapper::bmp_ (*C++ member*), 1200
 hpx::threads::hpx_hwloc_bitmap_wrapper::get_bmp

(*C++ function*), 1200
hpx::threads::hpx_hwloc_bitmap_wrapper::hpx_hwloc_bitmap_wrapper (*C++ function*), 1199
hpx::threads::hpx_hwloc_bitmap_wrapper::hpx_hwloc_bitmap_wrapper (*C++ function*), 1199
hpx::threads::hpx_hwloc_bitmap_wrapper::operator (C++ function), 1175
 bool (C++ function), 1199
hpx::threads::hpx_hwloc_bitmap_wrapper::operator<< (C++ function), 1179, 1180
 (C++ function), 1200
hpx::threads::hpx_hwloc_bitmap_wrapper::reset hpx::threads::suspend_pool (C++ function), 1157
 (C++ function), 1199
hpx::threads::hpx_hwloc_membind_policy (C++ enum), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_point (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_thread (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_thread_data (C++ class), 1170
 (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_thread_data (~thread_data)
hpx::threads::hpx_hwloc_membind_policy::membind_first (C++ function), 1173
 (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_interleave (C++ function), 1172
 (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_mixed (C++ member), 1174
 (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_nextto (C++ enumerator), 1173
 (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_replacement (C++ function), 1172
 (C++ enumerator), 1199
hpx::threads::hpx_hwloc_membind_policy::membind_user (C++ member), 1174
 (C++ enumerator), 1199
hpx::threads::hwloc_bitmap_ptr (C++ type), 1199
hpx::threads::interrupt_thread (C++ function), 1182
hpx::threads::interruption_point (C++ function), 1182
hpx::threads::invalid_thread_id (C++ member), 949
hpx::threads::make_thread_function (C++ function), 1167
hpx::threads::make_thread_function_nullary (C++ function), 1167
hpx::threads::operator<< (C++ function), 947, 948, 1185
hpx::threads::policies (C++ type), 1187
hpx::threads::register_thread (C++ function), 1167
hpx::threads::register_work (C++ function), 1167, 1168
hpx::threads::resume_pool (C++ function), 1157
hpx::threads::resume_pool_cb (C++ function), 1157
hpx::threads::resume_processing_unit (C++ function), 1156
hpx::threads::resume_processing_unit_cb (C++ function), 1156
hpx::threads::set_thread_description (C++ function), 1175
hpx::threads::set_thread_interruption_enabled (C++ function), 1181
hpx::threads::set_thread_lco_description (C++ function), 1175
hpx::threads::set_thread_state (C++ function), 1175
hpx::threads::suspend_pool (C++ function), 1157
hpx::threads::suspend_pool_cb (C++ function), 1157
hpx::threads::suspend_processing_unit (C++ function), 1156
hpx::threads::suspend_processing_unit_cb (C++ function), 1156
hpx::threads::thread_data (C++ class), 1170
hpx::threads::thread_data (~thread_data)
hpx::threads::thread_data::add_thread_exit_callback (C++ function), 1173
hpx::threads::thread_data::current_state (C++ function), 1174
hpx::threads::thread_data::destroy (C++ function), 1173
hpx::threads::thread_data::destroy_thread (C++ function), 1172
hpx::threads::thread_data::enabled_interrupt (C++ function), 1172
hpx::threads::thread_data::exit_funcs_ (C++ member), 1174
hpx::threads::thread_data::free_thread_exit_callbacks (C++ function), 1172
hpx::threads::thread_data::get_backtrace (C++ function), 1172
hpx::threads::thread_data::get_component_id (C++ function), 1172
hpx::threads::thread_data::get_description (C++ function), 1172
hpx::threads::thread_data::get_last_worker_thread_num (C++ function), 1173
hpx::threads::thread_data::get_lco_description (C++ function), 1172
hpx::threads::thread_data::get_parent_locality_id (C++ function), 1172
hpx::threads::thread_data::get_parent_thread_id (C++ function), 1172
hpx::threads::thread_data::get_parent_thread_phase (C++ function), 1172
hpx::threads::thread_data::get_priority (C++ function), 1172
hpx::threads::thread_data::get_queue (C++ function), 1173
hpx::threads::thread_data::get_scheduler_base (C++ function), 1172

<code>hpx::threads::thread_data::get_stack_size</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_data::set_interruption_enabled</code>	<code>(C++ function), 1172</code>
<code>hpx::threads::thread_data::get_stack_size_enum</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_data::set_last_worker_thread_num</code>	<code>(C++ function), 1173</code>
<code>hpx::threads::thread_data::get_state</code>	<code>(C++ function), 1171</code>	<code>hpx::threads::thread_data::set_lco_description</code>	<code>(C++ function), 1172</code>
<code>hpx::threads::thread_data::get_thread_data</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_data::set_priority</code>	<code>(C++ function), 1172</code>
<code>hpx::threads::thread_data::get_thread_id</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_data::set_state</code>	<code>(C++ function), 1171</code>
<code>hpx::threads::thread_data::get_thread_phase</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_data::set_state_ex</code>	<code>(C++ function), 1173</code>
<code>hpx::threads::thread_data::init</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_data::set_state_tagged</code>	<code>(C++ function), 1171</code>
<code>hpx::threads::thread_data::interrupt</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::thread_data::set_thread_data</code>	<code>(C++ function), 1173</code>
<code>hpx::threads::thread_data::interruption_enable</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::spinlock_pool</code>	<code>(C++ type), 1171</code>
<code>hpx::threads::thread_data::interruption_point</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::stacksize_</code>	<code>(C++ member), 1174</code>
<code>hpx::threads::thread_data::interruption_requested</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::thread_data</code>	<code>(C++ function), 1171, 1173</code>
<code>hpx::threads::thread_data::is_stackless</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::thread_id</code>	<code>(C++ struct), 949</code>
<code>hpx::threads::thread_data::is_stackless_</code>	<code>(C++ member), 1174</code>	<code>hpx::threads::thread_id::format_value</code>	<code>(C++ function), 951</code>
<code>hpx::threads::thread_data::last_worker_thread_num_</code>	<code>(C++ member), 1174</code>	<code>hpx::threads::thread_id::get</code>	<code>(C++ function), 950</code>
<code>hpx::threads::thread_data::operator()</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_id::operator bool</code>	<code>(C++ function), 950</code>
<code>hpx::threads::thread_data::operator=</code>	<code>(C++ function), 1171</code>	<code>hpx::threads::thread_id::operator=</code>	<code>(C++ function), 950</code>
<code>hpx::threads::thread_data::priority_</code>	<code>(C++ member), 1174</code>	<code>hpx::threads::thread_id::operator<<</code>	<code>(C++ function), 951</code>
<code>hpx::threads::thread_data::queue_</code>	<code>(C++ member), 1174</code>	<code>hpx::threads::thread_id::reset</code>	<code>(C++ function), 950</code>
<code>hpx::threads::thread_data::ran_exit_funcs_</code>	<code>(C++ member), 1174</code>	<code>hpx::threads::thread_id::thrd_</code>	<code>(C++ member), 950</code>
<code>hpx::threads::thread_data::rebind</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_id::thread_id</code>	<code>(C++ function), 950</code>
<code>hpx::threads::thread_data::rebind_base</code>	<code>(C++ function), 1173</code>	<code>hpx::threads::thread_id::thread_id_repr</code>	<code>(C++ type), 950</code>
<code>hpx::threads::thread_data::requested_interrupt</code>	<code>(C++ member), 1174</code>	<code>hpx::threads::thread_id::addr</code>	<code>(C++ enum), 949</code>
<code>hpx::threads::thread_data::restore_state</code>	<code>(C++ function), 1171, 1172</code>	<code>hpx::threads::thread_id::addr::no</code>	<code>(C++ enumerator), 949</code>
<code>hpx::threads::thread_data::run_thread_exit_callbacks</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::thread_id::addr::yes</code>	<code>(C++ enumerator), 949</code>
<code>hpx::threads::thread_data::scheduler_base_</code>	<code>(C++ member), 1174</code>	<code>hpx::threads::thread_id::ref</code>	<code>(C++ struct), 951</code>
<code>hpx::threads::thread_data::set_backtrace</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::thread_id::ref::detach</code>	<code>(C++ function), 952</code>
<code>hpx::threads::thread_data::set_description</code>	<code>(C++ function), 1172</code>	<code>hpx::threads::thread_id::ref::format_value</code>	<code>(C++ function), 953</code>
		<code>hpx::threads::thread_id::ref::get</code>	<code>(C++ function), 952</code>

hpx::threads::thread_id_ref::noref (C++ function), 952
hpx::threads::thread_id_ref::operator bool (C++ function), 952
hpx::threads::thread_id_ref::operator!= (C++ function), 952
hpx::threads::thread_id_ref::operator= (C++ function), 951, 952
hpx::threads::thread_id_ref::operator== (C++ function), 952
hpx::threads::thread_id_ref::operator> (C++ function), 952
hpx::threads::thread_id_ref::operator>= (C++ function), 952
hpx::threads::thread_id_ref::operator< (C++ function), 952
hpx::threads::thread_id_ref::operator<= (C++ function), 952
hpx::threads::thread_id_ref::operator<< (C++ function), 952
hpx::threads::thread_id_ref::reset (C++ function), 952
hpx::threads::thread_id_ref::thrd_ (C++ member), 952
hpx::threads::thread_id_ref::thread_id_ref (C++ function), 951
hpx::threads::thread_id_ref::thread_id_repr (C++ type), 952
hpx::threads::thread_id_ref::thread_repr (C++ type), 951
hpx::threads::thread_id_ref_type (C++ type), 1141
hpx::threads::thread_pool_base (C++ class), 1185
hpx::threads::thread_pool_base::resume_direct (C++ function), 1186
hpx::threads::thread_pool_base::resume_processing_unit (C++ function), 1185
hpx::threads::thread_pool_base::suspend_direct (C++ function), 1186
hpx::threads::thread_pool_base::suspend_processing_unit (C++ function), 1185
hpx::threads::thread_pool_init_parameters (C++ struct), 1186
hpx::threads::thread_pool_init_parameters::affinity_data (C++ member), 1187
hpx::threads::thread_pool_init_parameters::index_ (C++ member), 1186
hpx::threads::thread_pool_init_parameters::max_background_threads (C++ member), 1187
hpx::threads::thread_pool_init_parameters::max_busy_lanes (C++ member), 1187
hpx::threads::thread_pool_init_parameters::max_idle_lanes (C++ member), 1187
hpx::threads::thread_pool_init_parameters::mode_ (C++ member), 1186
hpx::threads::thread_pool_init_parameters::name_ (C++ member), 1186
hpx::threads::thread_pool_init_parameters::network_backgr_ (C++ member), 1187
hpx::threads::thread_pool_init_parameters::notifier_ (C++ member), 1187
hpx::threads::thread_pool_init_parameters::num_threads_ (C++ member), 1186
hpx::threads::thread_pool_init_parameters::shutdown_check_ (C++ member), 1187
hpx::threads::thread_pool_init_parameters::thread_offset_ (C++ member), 1186
hpx::threads::thread_pool_init_parameters::thread_pool_init_ (C++ function), 1186
hpx::threads::thread_priority (C++ enum), 945
hpx::threads::thread_priority::boost (C++ enumerator), 945
hpx::threads::thread_priority::bound (C++ enumerator), 946
hpx::threads::thread_priority::default_ (C++ enumerator), 945
hpx::threads::thread_priority::high (C++ enumerator), 945
hpx::threads::thread_priority::high_recursive (C++ enumerator), 945
hpx::threads::thread_priority::low (C++ enumerator), 945
hpx::threads::thread_priority::normal (C++ enumerator), 945
hpx::threads::thread_priority::unknown (C++ enumerator), 945
hpx::threads::thread_restart_state (C++ enum), 946
hpx::threads::thread_restart_state::abort (C++ enumerator), 946
hpx::threads::thread_restart_state::signaled (C++ enumerator), 946
hpx::threads::thread_restart_state::terminate (C++ enumerator), 946
hpx::threads::thread_restart_state::timeout (C++ enumerator), 946
hpx::threads::thread_restart_state::unknown (C++ enumerator), 946
hpx::threads::thread_schedule_hint (C++ struct), 948
hpx::threads::thread_schedule_hint::hint (C++ member), 948
hpx::threads::thread_schedule_hint::mode (C++ member), 948
hpx::threads::thread_schedule_hint::thread_schedule_hint (C++ function), 948
hpx::threads::thread_schedule_hint::thread_schedule_hint_mode (C++ member), 948

```

(C++ enum), 947
hpx::threads::thread_schedule_hint_mode::none      (C++ function), 1190
hpx::threads::thread_schedule_hint_mode::numa       (C++ function), 1190
hpx::threads::thread_schedule_hint_mode::thread     (C++ function), 1189
hpx::threads::thread_schedule_state    (C++ enum), 944
hpx::threads::thread_schedule_state::active        (C++ enumerator), 944
hpx::threads::thread_schedule_state::depleted      (C++ enumerator), 945
hpx::threads::thread_schedule_state::pending        (C++ enumerator), 944
hpx::threads::thread_schedule_state::pending_boost  (C++ function), 1189
hpx::threads::thread_schedule_state::pending_do_not_schedule (C++ function), 1189
hpx::threads::thread_schedule_state::staged         (C++ enumerator), 945
hpx::threads::thread_schedule_state::suspended      (C++ enumerator), 945
hpx::threads::thread_schedule_state::terminated     (C++ enumerator), 945
hpx::threads::thread_schedule_state::unknown        (C++ enumerator), 944
hpx::threads::thread_self (C++ type), 1141
hpx::threads::thread_stacksize (C++ enum), 946
hpx::threads::thread_stacksize::current            (C++ enumerator), 947
hpx::threads::thread_stacksize::default_           (C++ enumerator), 947
hpx::threads::thread_stacksize::huge               (C++ enumerator), 946
hpx::threads::thread_stacksize::large              (C++ enumerator), 946
hpx::threads::thread_stacksize::maximal            (C++ enumerator), 947
hpx::threads::thread_stacksize::medium             (C++ enumerator), 946
hpx::threads::thread_stacksize::minimal            (C++ enumerator), 947
hpx::threads::thread_stacksize::nostack           (C++ enumerator), 946
hpx::threads::thread_stacksize::small_             (C++ enumerator), 946
hpx::threads::thread_stacksize::unknown            (C++ enumerator), 946
hpx::threads::threadmanager (C++ class), 1187
hpx::threads::threadmanager::~threadmanager      (C++ function), 1188
hpx::threads::threadmanager::abort_all_suspend    (C++ function), 1189
hpx::threads::threadmanager::add_remove_scheduler_mode (C++ function), 1190
hpx::threads::threadmanager::add_scheduler_mode   (C++ function), 1190
hpx::threads::threadmanager::cleanup_terminated   (C++ function), 1189
hpx::threads::threadmanager::create_pools         (C++ function), 1188
hpx::threads::threadmanager::default_pool         (C++ function), 1188
hpx::threads::threadmanager::default_scheduler    (C++ function), 1188
hpx::threads::threadmanager::deinit_tss           (C++ function), 1190
hpx::threads::threadmanager::enumerate_threads    (C++ function), 1189
hpx::threads::threadmanager::get_background_thread_count (C++ function), 1189
hpx::threads::threadmanager::get_cumulative_duration (C++ function), 1190
hpx::threads::threadmanager::get_idle_core_count  (C++ function), 1189
hpx::threads::threadmanager::get_idle_core_mask   (C++ function), 1189
hpx::threads::threadmanager::get_os_thread_count  (C++ function), 1190
hpx::threads::threadmanager::get_os_thread_handle (C++ function), 1190
hpx::threads::threadmanager::get_pool   (C++ function), 1188
hpx::threads::threadmanager::get_pool_numa_bitmap (C++ function), 1190
hpx::threads::threadmanager::get_queue_length     (C++ function), 1190
hpx::threads::threadmanager::get_thread_count     (C++ function), 1189
hpx::threads::threadmanager::get_thread_count_active (C++ function), 1190
hpx::threads::threadmanager::get_thread_count_pending (C++ function), 1190
hpx::threads::threadmanager::get_thread_count_staged (C++ function), 1190
hpx::threads::threadmanager::get_thread_count_suspended (C++ function), 1190
hpx::threads::threadmanager::get_thread_count_terminated (C++ function), 1190
hpx::threads::threadmanager::get_thread_count_unknown (C++ function), 1190
hpx::threads::threadmanager::get_used_processing_units (C++ function), 1190
hpx::threads::threadmanager::init (C++ function), 1188
hpx::threads::threadmanager::init_tss  (C++ function), 1190

```

hpx::threads::threadmanager::is_busy (C++ function), 1189
hpx::threads::threadmanager::is_idle (C++ function), 1189
hpx::threads::threadmanager::mtx_ (C++ member), 1191
hpx::threads::threadmanager::mutex_type (C++ type), 1191
hpx::threads::threadmanager::network_background (C++ member), 1191
hpx::threads::threadmanager::notification_policy (C++ type), 1188
hpx::threads::threadmanager::notifier_ (C++ member), 1191
hpx::threads::threadmanager::pool_exists (C++ function), 1188
hpx::threads::threadmanager::pool_type (C++ type), 1188
hpx::threads::threadmanager::pool_vector (C++ type), 1188
hpx::threads::threadmanager::pools_ (C++ member), 1191
hpx::threads::threadmanager::print_pools (C++ function), 1188
hpx::threads::threadmanager::register_thread (C++ function), 1188
hpx::threads::threadmanager::register_work (C++ function), 1188
hpx::threads::threadmanager::remove_scheduler (C++ function), 1190
hpx::threads::threadmanager::report_error (C++ function), 1190
hpx::threads::threadmanager::reset_thread (C++ function), 1190
hpx::threads::threadmanager::resume (C++ function), 1189
hpx::threads::threadmanager::rtcfg_ (C++ member), 1191
hpx::threads::threadmanager::run (C++ function), 1189
hpx::threads::threadmanager::scheduler_type (C++ type), 1188
hpx::threads::threadmanager::set_scheduler_mode (C++ function), 1190
hpx::threads::threadmanager::status (C++ function), 1189
hpx::threads::threadmanager::stop (C++ function), 1189
hpx::threads::threadmanager::suspend (C++ function), 1189
hpx::threads::threadmanager::threadmanager (C++ function), 1188
hpx::threads::threadmanager::threads_lookup_ (C++ member), 1191
hpx::threads::threadmanager::wait (C++ function), 1189
hpx::threads::topology (C++ struct), 1200
hpx::threads::topology::~topology (C++ function), 1200
hpx::threads::topology::allocate (C++ function), 1202
hpx::threads::topology::allocate_membind (C++ function), 1202
hpx::threads::topology::bitmap_to_mask (C++ function), 1203
hpx::threads::topology::core_affinity_masks_ (C++ member), 1204
hpx::threads::topology::core_numbers_ (C++ member), 1204
hpx::threads::topology::core_offset (C++ member), 1204
hpx::threads::topology::cpuset_to_nodeset (C++ function), 1202
hpx::threads::topology::deallocate (C++ function), 1202
hpx::threads::topology::empty_mask (C++ member), 1204
hpx::threads::topology::extract_node_count (C++ function), 1203
hpx::threads::topology::extract_node_count_locked (C++ function), 1203
hpx::threads::topology::extract_node_mask (C++ function), 1203
hpx::threads::topology::get_area_membind_nodeset (C++ function), 1202
hpx::threads::topology::get_cache_size (C++ distribution function), 1202
hpx::threads::topology::get_core_affinity_mask (C++ function), 1201
hpx::threads::topology::get_core_number (C++ function), 1202
hpx::threads::topology::get_cpubind_mask (C++ function), 1202
hpx::threads::topology::get_machine_affinity_mask (C++ function), 1200
hpx::threads::topology::get_memory_page_size (C++ function), 1205
hpx::threads::topology::get numa_domain (C++ function), 1202
hpx::threads::topology::get numa_node_affinity_mask (C++ function), 1200
hpx::threads::topology::get numa_node_affinity_mask_from_r (C++ function), 1201
hpx::threads::topology::get numa_node_number (C++ function), 1200
hpx::threads::topology::get number_of_core_pus (C++ function), 1202
hpx::threads::topology::get number_of_core_pus_locked

```

(C++ function), 1203
hpx::threads::topology::get_number_of_cores    hpx::threads::topology::init_thread_affinity_mask
(C++ function), 1202                           (C++ function), 1203
hpx::threads::topology::get_number_of numa_nodes hpx::threads::topology::machine_affinity_mask_
(C++ function), 1202                           (C++ member), 1204
hpx::threads::topology::get_number_of numa_nodes hpx::threads::topology::mask_to_bitmap (C++
(C++ function), 1202                           function), 1203
hpx::threads::topology::get_number_of numa_nodes hpx::threads::topology::memory_page_size_
(C++ function), 1201                           (C++ member), 1204
hpx::threads::topology::get_number_of pus      hpx::threads::topology::mutex_type (C++ type),
(C++ function), 1202                           1203
hpx::threads::topology::get_number_of socket_d pines:threads::topology::num_of_pus_     (C++
(C++ function), 1202                           member), 1204
hpx::threads::topology::get_number_of socket_p hpx::threads::topology::numa_node_affinity_masks_
(C++ function), 1202                           (C++ member), 1204
hpx::threads::topology::get_number_of sockets hpx::threads::topology::numa_node_numbers_
(C++ function), 1201                           (C++ member), 1204
hpx::threads::topology::get_pu_number   (C++ hpx::threads::topology::print_affinity_mask
function), 1202                           (C++ function), 1201
hpx::threads::topology::get_pu_obj (C++ func- hpx::threads::topology::print_hwloc     (C++
tion), 1203                           function), 1203
hpx::threads::topology::get_service_affinity_m hpx::threads::topology::print_mask_vector
ask:threads::topology::print_vector     (C++
(C++ function), 1200                           function), 1203
hpx::threads::topology::get_socket_affinity_m hpx::threads::topology::print_vector     (C++
ask,threads::topology::set_area_membind_nodeset
(C++ function), 1200                           (C++ function), 1202
hpx::threads::topology::init_core_affinity_m hpx::threads::topology::set_thread_affinity_mask
ask:threads::topology::set_thread_affinity_m
(C++ function), 1203                           (C++ function), 1201
hpx::threads::topology::init_core_affinity_m hpx::threads::topology::socket_affinity_masks_
ask:threads::topology::socket_numbers_
(C++ function), 1203                           (C++ member), 1204
hpx::threads::topology::init_core_number       hpx::threads::topology::socket_numbers_
(C++ function), 1203                           (C++ member), 1204
hpx::threads::topology::init_machine_affinity_hpx::threads::topology::thread_affinity_masks_
mask:threads::topology::topo (C++ member),
(C++ function), 1203                           1204
hpx::threads::topology::init_node_number       hpx::threads::topology::topo_mtx (C++ mem-
(C++ function), 1203                           ber), 1204
hpx::threads::topology::init_num_of_pus        hpx::threads::topology::topo_mtx (C++ mem-
(C++ function), 1203                           ber), 1204
hpx::threads::topology::init numa_node_affini hpx::threads::topology::topology (C++ func-
(C++ function), 1203                           tion), 1200
hpx::threads::topology::init numa_node_affini hpx::threads::topology::topology (C++ func-
(C++ function), 1203                           tion), 1200
hpx::threads::topology::init numa_node_number hpx::threads::topology::use_pus_as_cores_
(C++ function), 1203                           (C++ member), 1204
hpx::threads::topology::init socket_affinity_m hpx::threads::topology::write_to_log     (C++
ask:throwmode (C++ enum), 985
(C++ function), 1203                           hpx::threads::topology::lightweight (C++ enumerator),
hpx::threads::topology::init socket_affinity_m hpx::threads::topology::plain (C++ enumerator), 985
ask:throwmode (C++ enum), 985
(C++ function), 1203                           hpx::threads::topology::rethrow (C++ enumerator), 985
hpx::threads::topology::init socket_number      hpx::threads::topology::rethrow (C++ enumerator), 985

```

hpx::throws (*C++ member*), 985
hpx::tie (*C++ function*), 967
hpx::timed_mutex (*C++ class*), 1138
hpx::timed_mutex::~timed_mutex (*C++ function*),
 1138
hpx::timed_mutex::HPX_NON_COPYABLE (*C++ func-
tion*), 1138
hpx::timed_mutex::lock (*C++ function*), 1140
hpx::timed_mutex::timed_mutex (*C++ function*),
 1138
hpx::timed_mutex::try_lock (*C++ function*), 1140,
 1141
hpx::timed_mutex::try_lock_for (*C++ function*),
 1139
hpx::timed_mutex::try_lock_until (*C++ func-
tion*), 1138
hpx::timed_mutex::unlock (*C++ function*), 1141
hpx::tolerate_node_faults (*C++ function*), 1099
hpx::traits (*C++ type*), 1013, 1031, 1032, 1042, 1043,
 1166, 1196, 1214, 1243
hpx::traits::action_remote_result (*C++ struct*),
 1214
hpx::traits::action_remote_result_t (*C++
type*), 1214
hpx::traits::instead (*C++ type*), 1042
hpx::traits::is_executor_parameters (*C++
struct*), 1013
hpx::traits::is_executor_parameters_t (*C++
type*), 1013
hpx::traits::is_executor_parameters_v (*C++
member*), 1013
hpx::traits::is_timed_executor (*C++ struct*),
 1196
hpx::transform (*C++ function*), 468–471
hpx::transform_exclusive_scan (*C++ function*),
 472, 473
hpx::transform_inclusive_scan (*C++ function*),
 475–477, 479
hpx::transform_reduce (*C++ function*), 480, 482–
 484, 486, 831, 833–836, 838–844
hpx::trigger_lco_event (*C++ function*), 1250
hpx::tuple (*C++ class*), 967
hpx::tuple_cat (*C++ function*), 967
hpx::tuple_element (*C++ struct*), 968
hpx::tuple_size (*C++ struct*), 968
hpx::uninitialized_copy (*C++ function*), 487
hpx::uninitialized_copy_n (*C++ function*), 488,
 489
hpx::uninitialized_default_construct (*C++
function*), 490
hpx::uninitialized_default_construct_n (*C++
function*), 491, 492
hpx::uninitialized_fill (*C++ function*), 493
hpx::uninitialized_fill_n (*C++ function*), 494
hpx::uninitialized_move (*C++ function*), 496
hpx::uninitialized_move_n (*C++ function*), 497
hpx::uninitialized_value_construct (*C++ func-
tion*), 499
hpx::uninitialized_value_construct_n (*C++
function*), 500
hpx::unique (*C++ function*), 501, 502
hpx::unique_any_nonsr (*C++ type*), 958
hpx::unique_copy (*C++ function*), 503, 504
hpx::unregister_thread (*C++ function*), 1098
hpx::unregister_with_basename (*C++ function*),
 1298
hpx::unwrap (*C++ function*), 1067
hpx::unwrap_all (*C++ function*), 1068
hpx::unwrap_n (*C++ function*), 1068
hpx::unwrapping (*C++ function*), 1068
hpx::unwrapping_all (*C++ function*), 1068
hpx::unwrapping_n (*C++ function*), 1068
hpx::util (*C++ type*), 884, 913, 920, 924, 926, 928,
 929, 931–933, 960, 970, 1031, 1032, 1034,
 1036, 1042, 1055, 1064, 1065, 1098, 1166,
 1168, 1175, 1205, 1255, 1263
hpx::util::accept_begin (*C++ function*), 885
hpx::util::accept_end (*C++ function*), 885
hpx::util::annotated_function (*C++ function*),
 1166
hpx::util::as_string (*C++ function*), 1175
hpx::util::basic_any (*C++ class*), 961
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type> (*C++ class*), 968
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::~basic_any (*C++
function*), 969
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::assign (*C++ function*),
 969
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::basic_any (*C++
function*), 968
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::cast (*C++ function*),
 969
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::equal_to (*C++ func-
tion*), 969
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::has_value (*C++
function*), 969
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::HPX_SERIALIZATION_SPLIT_MEMBER
 (*C++ function*), 969
hpx::util::basic_any<IArch, OArch, Char,
 std::true_type>::load (*C++ function*),
 969

```

hpx::util::basic_any<IArch, OArch, Char,
    std::true_type>::new_object      (C++  

        function), 969
hpx::util::basic_any<IArch, OArch, Char,  

    std::true_type>::object (C++ member),  

    969
hpx::util::basic_any<IArch, OArch, Char,  

    std::true_type>::operator=      (C++  

        function), 969
hpx::util::basic_any<IArch, OArch, Char,  

    std::true_type>::reset (C++ function),  

    969
hpx::util::basic_any<IArch, OArch, Char,  

    std::true_type>::save (C++ function),  

    969
hpx::util::basic_any<IArch, OArch, Char,  

    std::true_type>::swap (C++ function),  

    969
hpx::util::basic_any<IArch, OArch, Char,  

    std::true_type>::table (C++ member),  

    969
hpx::util::basic_any<IArch, OArch, Char,  

    std::true_type>::type (C++ function),  

    969
hpx::util::basic_any<void, void, Char,  

    std::false_type>(C++ class), 957
hpx::util::basic_any<void, void, Char,  

    std::false_type>::~basic_any     (C++  

        function), 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::basic_any     (C++  

        function), 957
hpx::util::basic_any<void, void, Char,  

    std::false_type>::cast (C++ function),  

    958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::equal_to      (C++  

        function), 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::has_value     (C++  

        function), 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::new_object    (C++  

        function), 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::object (C++ mem-  

        ber), 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::operator=     (C++  

        function), 957, 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::reset (C++ func-  

        tion), 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::swap (C++ function),  

    958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::table (C++ mem-  

        ber), 958
hpx::util::basic_any<void, void, Char,  

    std::false_type>::type (C++ function),  

    958
hpx::util::basic_any<void, void, Char,  

    std::true_type>(C++ class), 954
hpx::util::basic_any<void, void, Char,  

    std::true_type>::~basic_any     (C++  

        function), 955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::assign (C++ func-  

        tion), 955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::basic_any     (C++  

        function), 955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::cast (C++ function),  

    955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::equal_to      (C++ func-  

        tion), 955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::has_value     (C++  

        function), 955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::new_object    (C++  

        function), 956
hpx::util::basic_any<void, void, Char,  

    std::true_type>::object (C++ mem-  

        ber), 956
hpx::util::basic_any<void, void, Char,  

    std::true_type>::operator=     (C++  

        function), 955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::reset (C++ function),  

    955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::swap (C++ function),  

    955
hpx::util::basic_any<void, void, Char,  

    std::true_type>::table (C++ mem-  

        ber), 956
hpx::util::basic_any<void, void, Char,  

    std::true_type>::type (C++ function),  

    955
hpx::util::basic_any<void, void, void,  

    std::false_type>(C++ class), 956
hpx::util::basic_any<void, void, void,  

    std::false_type>::~basic_any     (C++  

        function), 956

```

hpx::util::basic_any<void, void, void,
std::false_type>::basic_any (C++
function), 956

hpx::util::basic_any<void, void, void,
std::false_type>::cast (C++ function),
957

hpx::util::basic_any<void, void, void,
std::false_type>::equal_to (C++
function), 957

hpx::util::basic_any<void, void, void,
std::false_type>::has_value (C++
function), 957

hpx::util::basic_any<void, void, void,
std::false_type>::new_object (C++
function), 957

hpx::util::basic_any<void, void, void,
std::false_type>::object (C++ mem-
ber), 957

hpx::util::basic_any<void, void, void,
std::false_type>::operator= (C++
function), 956

hpx::util::basic_any<void, void, void,
std::false_type>::reset (C++ func-
tion), 957

hpx::util::basic_any<void, void, void,
std::false_type>::swap (C++ function),
956

hpx::util::basic_any<void, void, void,
std::false_type>::table (C++ mem-
ber), 957

hpx::util::basic_any<void, void, void,
std::false_type>::type (C++ function),
956

hpx::util::basic_any<void, void, void,
std::true_type> (C++ class), 953

hpx::util::basic_any<void, void, void,
std::true_type>::~basic_any (C++
function), 954

hpx::util::basic_any<void, void, void,
std::true_type>::assign (C++ func-
tion), 954

hpx::util::basic_any<void, void, void,
std::true_type>::basic_any (C++
function), 953, 954

hpx::util::basic_any<void, void, void,
std::true_type>::cast (C++ function),
954

hpx::util::basic_any<void, void, void,
std::true_type>::equal_to (C++ func-
tion), 954

hpx::util::basic_any<void, void, void,
std::true_type>::has_value (C++
function), 954

hpx::util::basic_any<void, void, void,
std::true_type>::new_object (C++
function), 954

hpx::util::basic_any<void, void, void,
std::true_type>::object (C++ mem-
ber), 954

hpx::util::basic_any<void, void, void,
std::true_type>::operator= (C++
function), 954

hpx::util::basic_any<void, void, void,
std::true_type>::reset (C++ function),
954

hpx::util::basic_any<void, void, void,
std::true_type>::swap (C++ function),
954

hpx::util::basic_any<void, void, void,
std::true_type>::table (C++ mem-
ber), 954

hpx::util::basic_any<void, void, void,
std::true_type>::type (C++ function),
954

hpx::util::cache (C++ type), 913, 920, 924, 926,
928, 929, 931–933

hpx::util::cache::entries (C++ type), 924, 926,
928, 929, 931

hpx::util::cache::entries::entry (C++ class),
924

hpx::util::cache::entries::entry::entry
(C++ function), 925

hpx::util::cache::entries::entry::get (C++
function), 926

hpx::util::cache::entries::entry::get_size
(C++ function), 925

hpx::util::cache::entries::entry::insert
(C++ function), 925

hpx::util::cache::entries::entry::operator<
(C++ function), 926

hpx::util::cache::entries::entry::remove
(C++ function), 925

hpx::util::cache::entries::entry::touch
(C++ function), 925

hpx::util::cache::entries::entry::value_
(C++ member), 926

hpx::util::cache::entries::entry::value_type
(C++ type), 925

hpx::util::cache::entries::fifo_entry (C++
class), 926

hpx::util::cache::entries::fifo_entry::base_type
(C++ type), 927

hpx::util::cache::entries::fifo_entry::fifo_entry
(C++ function), 927

hpx::util::cache::entries::fifo_entry::get_creation_time
(C++ function), 927

hpx::util::cache::entries::fifo_entry::insert
(C++ function), 927

```

hpx::util::cache::entries::fifo_entry::insertion_time(C++ member), 920
    (C++ member), 927                                hpx::util::cache::local_cache::adapt::operator()
hpx::util::cache::entries::fifo_entry::operator<      (C++ function), 920
    (C++ function), 927                                hpx::util::cache::local_cache::adapted_update_policy_type
hpx::util::cache::entries::fifo_entry::time_point      (C++ type), 919
    (C++ type), 927                                hpx::util::cache::local_cache::capacity
hpx::util::cache::entries::lfu_entry     (C++ class), 928   (C++ function), 915
hpx::util::cache::entries::lfu_entry::base_type        (C++ type), 929
hpx::util::cache::entries::lfu_entry::base_type        (C++ type), 929                                hpx::util::cache::local_cache::const_iterator
hpx::util::cache::entries::lfu_entry::get_access_count(C++ type), 919
    (C++ function), 928                                hpx::util::cache::local_cache::current_size_
hpx::util::cache::entries::lfu_entry::lfu_entry        (C++ function), 928
    (C++ function), 928                                hpx::util::cache::local_cache::entry_heap_
hpx::util::cache::entries::lfu_entry::operator<        (C++ member), 919
    (C++ function), 929                                hpx::util::cache::local_cache::entry_type
hpx::util::cache::entries::lfu_entry::ref_count_       (C++ type), 914
    (C++ member), 929                                hpx::util::cache::local_cache::erase   (C++ function), 918
hpx::util::cache::entries::lfu_entry::touch           (C++ function), 928
hpx::util::cache::entries::lru_entry     (C++ class), 929   hpx::util::cache::local_cache::free_space
hpx::util::cache::entries::lru_entry::access_time_    (C++ function), 915, 916
    (C++ member), 930                                hpx::util::cache::local_cache::get_statistics
hpx::util::cache::entries::lru_entry::base_type        (C++ type), 918
    (C++ type), 930                                hpx::util::cache::local_cache::heap_iterator
hpx::util::cache::entries::lru_entry::get_access_time(C++ type), 919
    (C++ function), 930                                hpx::util::cache::local_cache::heap_type
hpx::util::cache::entries::lru_entry::lru_entry        (C++ function), 930
    (C++ function), 930                                hpx::util::cache::local_cache::holds_key
hpx::util::cache::entries::lru_entry::operator<        (C++ function), 915
    (C++ function), 930                                hpx::util::cache::local_cache::insert   (C++ function), 916
hpx::util::cache::entries::lru_entry::time_point       (C++ type), 930
    (C++ type), 930                                hpx::util::cache::local_cache::insert_policy_
hpx::util::cache::entries::lru_entry::touch           (C++ function), 930
    (C++ function), 930                                hpx::util::cache::local_cache::insert_policy_type
hpx::util::cache::entries::size_entry   (C++ class), 931   (C++ type), 914
hpx::util::cache::entries::size_entry::base_type        (C++ type), 919
    (C++ type), 932                                hpx::util::cache::local_cache::iterator
hpx::util::cache::entries::size_entry::derived_type    (C++ type), 914
    (C++ type), 932                                hpx::util::cache::local_cache::local_cache
hpx::util::cache::entries::size_entry::get_size         (C++ function), 914
    (C++ function), 931                                hpx::util::cache::local_cache::max_size_
hpx::util::cache::entries::size_entry::size_            (C++ member), 932
    (C++ member), 932                                hpx::util::cache::local_cache::reserve  (C++ function), 915
hpx::util::cache::entries::size_entry::size_entry       (C++ function), 931
    (C++ function), 931                                hpx::util::cache::local_cache::size    (C++ function), 915
hpx::util::cache::local_cache (C++ class), 913
hpx::util::cache::local_cache::adapt   (C++ struct), 919
hpx::util::cache::local_cache::adapt::adapt  (C++ function), 920
hpx::util::cache::local_cache::adapt::f_

```

(C++ type), 914
hpx::util::cache::local_cache::storage_type (C++ type), 914
hpx::util::cache::local_cache::storage_value_type (C++ type), 914
hpx::util::cache::local_cache::store_ (C++ member), 919
hpx::util::cache::local_cache::update (C++ function), 917
hpx::util::cache::local_cache::update_if (C++ function), 917
hpx::util::cache::local_cache::update_on_exit (C++ type), 919
hpx::util::cache::local_cache::update_policy_ (C++ member), 919
hpx::util::cache::local_cache::update_policy_type (C++ type), 914
hpx::util::cache::local_cache::value_type (C++ type), 914
hpx::util::cache::lru_cache (C++ class), 920
hpx::util::cache::lru_cache::capacity (C++ function), 921
hpx::util::cache::lru_cache::clear (C++ function), 923
hpx::util::cache::lru_cache::current_size_ (C++ member), 924
hpx::util::cache::lru_cache::entry_pair (C++ type), 920
hpx::util::cache::lru_cache::entry_type (C++ type), 920
hpx::util::cache::lru_cache::erase (C++ function), 923
hpx::util::cache::lru_cache::evict (C++ function), 924
hpx::util::cache::lru_cache::get_entry (C++ function), 921, 922
hpx::util::cache::lru_cache::get_statistics (C++ function), 923
hpx::util::cache::lru_cache::holds_key (C++ function), 921
hpx::util::cache::lru_cache::insert (C++ function), 922
hpx::util::cache::lru_cache::insert_nonexist (C++ function), 924
hpx::util::cache::lru_cache::key_type (C++ type), 920
hpx::util::cache::lru_cache::lru_cache (C++ function), 921
hpx::util::cache::lru_cache::map_ (C++ member), 924
hpx::util::cache::lru_cache::map_type (C++ type), 920
hpx::util::cache::lru_cache::max_size_ (C++ member), 924
hpx::util::cache::lru_cache::reserve (C++ function), 921
hpx::util::cache::lru_cache::size (C++ function), 921
hpx::util::cache::lru_cache::size_type (C++ type), 921
hpx::util::cache::lru_cache::statistics_ (C++ member), 924
hpx::util::cache::lru_cache::statistics_type (C++ type), 920
hpx::util::cache::lru_cache::storage_ (C++ member), 924
hpx::util::cache::lru_cache::storage_type (C++ type), 920
hpx::util::cache::lru_cache::touch (C++ function), 924
hpx::util::cache::lru_cache::update (C++ function), 922
hpx::util::cache::lru_cache::update_if (C++ function), 922
hpx::util::cache::lru_cache::update_on_exit (C++ type), 924
hpx::util::cache::statistics (C++ type), 932, 933
hpx::util::cache::statistics::local_statistics (C++ class), 932
hpx::util::cache::statistics::local_statistics::clear (C++ function), 933
hpx::util::cache::statistics::local_statistics::evictions (C++ function), 932, 933
hpx::util::cache::statistics::local_statistics::evictions_ (C++ member), 933
hpx::util::cache::statistics::local_statistics::get_and_re (C++ function), 932
hpx::util::cache::statistics::local_statistics::got_evicti (C++ function), 933
hpx::util::cache::statistics::local_statistics::got_hit (C++ function), 933
hpx::util::cache::statistics::local_statistics::got_insert (C++ function), 933
hpx::util::cache::statistics::local_statistics::got_miss (C++ function), 933
hpx::util::cache::statistics::local_statistics::hits (C++ function), 932
hpx::util::cache::statistics::local_statistics::hits_ (C++ member), 933
hpx::util::cache::statistics::local_statistics::insertions (C++ function), 932, 933
hpx::util::cache::statistics::local_statistics::insertions_ (C++ member), 933
hpx::util::cache::statistics::local_statistics::local_stat (C++ function), 932
hpx::util::cache::statistics::local_statistics::misses (C++ function), 932

```

hpx::util::cache::statistics::local_statistics::missession), 1261
    (C++ member), 933                                     hpx::util::checkpoint::begin (C++ function),
hpx::util::cache::statistics::method   (C++           1261
    enum), 934                                         hpx::util::checkpoint::checkpoint (C++ func-
hpx::util::cache::statistics::method::erase_entry      tion), 1261
    (C++ enumerator), 934                               hpx::util::checkpoint::const_iterator (C+++
hpx::util::cache::statistics::method::get_entry        type), 1261
    (C++ enumerator), 934                               hpx::util::checkpoint::data (C++ function), 1261
hpx::util::cache::statistics::method::insert_entry     hpx::util::checkpoint::data_ (C++ member),
    (C++ enumerator), 934                               1261
hpx::util::cache::statistics::method::update_entry     hpx::util::checkpoint::end (C++ function), 1261
    (C++ enumerator), 934                               hpx::util::checkpoint::operator!= (C++ func-
hpx::util::cache::statistics::method_erase_entry       tion), 1262
    (C++ member), 934                                 hpx::util::checkpoint::operator= (C++ func-
hpx::util::cache::statistics::method_get_entry         tion), 1261
    (C++ member), 934                                 hpx::util::checkpoint::operator== (C++ func-
hpx::util::cache::statistics::method_insert_entry       1262
    (C++ member), 934                               hpx::util::checkpoint::operator>> (C++ func-
hpx::util::cache::statistics::method_update_entry       1262
    (C++ member), 934                               hpx::util::checkpoint::operator<< (C++ func-
hpx::util::cache::statistics::no_statistics            tion), 1261
    (C++ class), 934                                hpx::util::checkpoint::restore_checkpoint
hpx::util::cache::statistics::no_statistics::clear     (C++ function), 1262
    (C++ function), 934                               hpx::util::checkpoint::serialize (C++ func-
hpx::util::cache::statistics::no_statistics::get_eraseentry1261
    (C++ function), 935                               hpx::util::checkpoint::size (C++ function), 1261
hpx::util::cache::statistics::no_statistics::get_eraseentry1261
    (C++ function), 935                               hpx::util::clean_ip_address (C++ function),
hpx::util::cache::statistics::no_statistics::get_get_entry_count
    (C++ function), 934                               hpx::util::connect_begin (C++ function), 885
hpx::util::cache::statistics::no_statistics::get_get_entry_count
    (C++ function), 935                               hpx::util::endpoint_end (C++ function), 885
hpx::util::cache::statistics::no_statistics::get_insertentry884
    (C++ function), 935                               hpx::util::endpoint_iterator_type (C++ type),
hpx::util::cache::statistics::no_statistics::get_insertentry884
    (C++ function), 935                               hpx::util::get_endpoint (C++ function), 885
hpx::util::cache::statistics::no_statistics::get_insertentry884
    (C++ function), 935                               hpx::util::hash_any (C++ struct), 972
hpx::util::cache::statistics::no_statistics::get_insertentry884
    (C++ function), 935                               hpx::util::io_service_pool::~io_service_pool
hpx::util::cache::statistics::no_statistics::got_hit     (C++ function), 1055
    (C++ function), 934                               hpx::util::io_service_pool::clear (C++ func-
hpx::util::cache::statistics::no_statistics::got_insertion1056
    (C++ function), 934                               hpx::util::io_service_pool::clear_locked
hpx::util::cache::statistics::no_statistics::got_miss    1056
    (C++ function), 934                               hpx::util::io_service_pool::continue_barrier_
hpx::util::cache::statistics::no_statistics::update_onexitmember, 1057
    (C++ struct), 935                               hpx::util::io_service_pool::get_io_service
hpx::util::cache::statistics::no_statistics::update_onexitfiniupdate1056_exit
    (C++ function), 935                               hpx::util::io_service_pool::get_name (C++ func-
hpx::util::checkpoint (C++ class), 1260
hpx::util::checkpoint::~checkpoint (C++ func-    hpx::util::io_service_pool::get_os_thread_handle

```

(*C++ function*), 1056
hpx::util::io_service_pool::HPX_NON_COPYABLE (*C++ function*), 1055
hpx::util::io_service_pool::init (*C++ function*), 1056
hpx::util::io_service_pool::initialize_work (*C++ function*), 1057
hpx::util::io_service_pool::io_service_pool (*C++ function*), 1055
hpx::util::io_service_pool::io_service_ptr (*C++ type*), 1057
hpx::util::io_service_pool::io_services_ (*C++ member*), 1057
hpx::util::io_service_pool::join (*C++ function*), 1056
hpx::util::io_service_pool::join_locked (*C++ function*), 1056
hpx::util::io_service_pool::mtx_ (*C++ member*), 1057
hpx::util::io_service_pool::next_io_service_ (*C++ member*), 1057
hpx::util::io_service_pool::notifier_ (*C++ member*), 1057
hpx::util::io_service_pool::pool_name_ (*C++ member*), 1057
hpx::util::io_service_pool::pool_name_postfix hpx::util::PhonyNameDueToError::has_value (*C++ member*), 1057
hpx::util::io_service_pool::pool_size_ (*C++ member*), 1057
hpx::util::io_service_pool::run (*C++ function*), 1055, 1056
hpx::util::io_service_pool::run_locked (*C++ function*), 1056
hpx::util::io_service_pool::size (*C++ function*), 1056
hpx::util::io_service_pool::stop (*C++ function*), 1056
hpx::util::io_service_pool::stop_locked (*C++ function*), 1056
hpx::util::io_service_pool::stopped (*C++ function*), 1056
hpx::util::io_service_pool::stopped_ (*C++ member*), 1057
hpx::util::io_service_pool::thread_run (*C++ function*), 1056
hpx::util::io_service_pool::threads_ (*C++ member*), 1057
hpx::util::io_service_pool::wait (*C++ function*), 1056
hpx::util::io_service_pool::wait_barrier_ (*C++ member*), 1057
hpx::util::io_service_pool::wait_locked (*C++ function*), 1056
hpx::util::io_service_pool::waiting_ (*C++ member*), 1057
hpx::util::io_service_pool::work_ (*C++ member*), 1057
hpx::util::make_any (*C++ function*), 970
hpx::util::make_streamable_any_nonservable (*C++ function*), 961
hpx::util::make_streamable_unique_any_nonservable (*C++ function*), 961
hpx::util::operator>> (*C++ function*), 961, 1255
hpx::util::operator<< (*C++ function*), 961, 1175, 1255
hpx::util::parse_sed_expression (*C++ function*), 1206
hpx::util::PhonyNameDueToError::~basic_any (*C++ function*), 962–965, 971
hpx::util::PhonyNameDueToError::assign (*C++ function*), 963, 966, 971
hpx::util::PhonyNameDueToError::basic_any (*C++ function*), 961–965, 971
hpx::util::PhonyNameDueToError::cast (*C++ function*), 962, 963, 965, 966, 971
hpx::util::PhonyNameDueToError::equal_to (*C++ function*), 962, 963, 965, 966, 971
hpx::util::PhonyNameDueToError::has_value (*C++ function*), 962, 963, 965, 966, 971
hpx::util::PhonyNameDueToError::HPX_SERIALIZATION_SPLIT_MEMBER (*C++ function*), 971
hpx::util::PhonyNameDueToError::load (*C++ function*), 971
hpx::util::PhonyNameDueToError::new_object (*C++ function*), 962, 964–966, 972
hpx::util::PhonyNameDueToError::object (*C++ member*), 962, 964–966, 972
hpx::util::PhonyNameDueToError::operator= (*C++ function*), 962–966, 971
hpx::util::PhonyNameDueToError::reset (*C++ function*), 962, 963, 965, 966, 971
hpx::util::PhonyNameDueToError::save (*C++ function*), 971
hpx::util::PhonyNameDueToError::swap (*C++ function*), 962–964, 966, 971
hpx::util::PhonyNameDueToError::table (*C++ member*), 962, 964–966, 972
hpx::util::PhonyNameDueToError::type (*C++ function*), 962–964, 966, 971
hpx::util::placeholders (*C++ type*), 1031
hpx::util::prepare_checkpoint (*C++ function*), 1258–1260
hpx::util::prepare_checkpoint_data (*C++ function*), 1263
hpx::util::resolve_hostname (*C++ function*), 885
hpx::util::resolve_public_ip_address (*C++ function*), 885

hpx::util::function), 885
 hpx::util::restore_checkpoint (C++ function), 1260
 hpx::util::restore_checkpoint_data (C++ function), 1263
 hpx::util::retrieve_commandline_arguments (C++ function), 1098
 hpx::util::save_checkpoint (C++ function), 1256–1258
 hpx::util::save_checkpoint_data (C++ function), 1263
 hpx::util::sed_transform (C++ struct), 1206
 hpx::util::sed_transform::command_ (C++ member), 1206
 hpx::util::sed_transform::operator bool (C++ function), 1206
 hpx::util::sed_transform::operator! (C++ function), 1206
 hpx::util::sed_transform::operator() (C++ function), 1206
 hpx::util::sed_transform::sed_transform (C++ function), 1206
 hpx::util::split_ip_address (C++ function), 885
 hpx::util::streamable_any_nonser (C++ type), 960
 hpx::util::streamable_unique_any_nonser (C++ type), 960
 hpx::util::streamable_unique_wany_nonser (C++ type), 960
 hpx::util::streamable_wany_nonser (C++ type), 960
 hpx::util::swap (C++ function), 961
 hpx::util::thread_description (C++ struct), 1175
 hpx::util::thread_description::data_type (C++ enum), 1175
 hpx::util::thread_description::data_type::data_type (C++ enumerator), 1175
 hpx::util::thread_description::data_type::data_type (C++ enumerator), 1175
 hpx::util::thread_description::get_address (C++ function), 1176
 hpx::util::thread_description::get_description (C++ function), 1176
 hpx::util::thread_description::init_from_alternative (C++ function), 1176
 hpx::util::thread_description::kind (C++ function), 1176
 hpx::util::thread_description::operator bool (C++ function), 1176
 hpx::util::thread_description::thread_descrip(C++ function), 1175, 1176
 hpx::util::thread_description::valid (C++ function), 1176

hpx::util::traverse_pack_async (C++ function), 1065
 hpx::util::traverse_pack_async_allocator (C++ function), 1066
 hpx::util::wany (C++ type), 970
 hpx::wait_all (C++ function), 893, 894
 hpx::wait_all_n (C++ function), 894
 hpx::wait_any (C++ function), 895, 896
 hpx::wait_any_n (C++ function), 896
 hpx::wait_each (C++ function), 897
 hpx::wait_each_n (C++ function), 898
 hpx::wait_some (C++ function), 899, 900
 hpx::wait_some_n (C++ function), 900
 hpx::when_all (C++ function), 901, 902
 hpx::when_all_n (C++ function), 902
 hpx::when_any (C++ function), 903, 904
 hpx::when_any_n (C++ function), 904
 hpx::when_any_result (C++ struct), 905
 hpx::when_any_result::futures (C++ member), 905
 hpx::when_any_result::index (C++ member), 905
 hpx::when_each (C++ function), 905, 906
 hpx::when_each_n (C++ function), 906
 hpx::when_some (C++ function), 907, 908
 hpx::when_some_n (C++ function), 909
 hpx::when_some_result (C++ struct), 909
 hpx::when_some_result::futures (C++ member), 910
 hpx::when_some_result::indices (C++ member), 910
 HPX_ASSERT (C macro), 887
 HPX_ASSERT_MSG (C macro), 887
 HPX_CACHE_METHOD_UNSCOPED_ENUM_DEPRECATED_MSG (C macro), 933
 HPX_COROUTINES_WITH_SWAP_CONTEXT_EMULATION:BOOL command line option, 56
 HPX_COUNTER_STATUS_UNSCOPED_ENUM_DEPRECATED_MSG (C macro), 1348
 HPX_COUNTER_TYPE_UNSCOPED_ENUM_DEPRECATED_MSG (C macro), 1348
 HPX_CURRENT_SOURCE_LOCATION (C macro), 886
 HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL command line option, 60
 HPX_DATASTRUCTURES_WITH_ADAPT_STD_VARIANT:BOOL command line option, 60
 HPX_DECLARE_PLAIN_ACTION (C macro), 1212
 HPX_DEFINE_COMPONENT_ACTION (C macro), 1210
 HPX_DEFINE_COMPONENT_COMMANDLINE_OPTIONS (C macro), 1306
 HPX_DEFINE_COMPONENT_NAME (C macro), 1308
 HPX_DEFINE_COMPONENT_NAME_ (C macro), 1308
 HPX_DEFINE_COMPONENT_NAME_2 (C macro), 1308
 HPX_DEFINE_COMPONENT_NAME_3 (C macro), 1308

HPX_DEFINE_COMPONENT_STARTUP_SHUTDOWN (*C macro*), 1307
HPX_DEFINE_GET_COMPONENT_TYPE (*C macro*), 1308
HPX_DEFINE_GET_COMPONENT_TYPE_STATIC (*C macro*), 1308
HPX_DEFINE_GET_COMPONENT_TYPE_TEMPLATE (*C macro*), 1308
HPX_DEFINE_PLAIN_ACTION (*C macro*), 1212
HPX_DISCOVER_COUNTERS_MODE_UNSCOPED_ENUM_DEPRECATED (*MSG macro*), 1348
HPX_DP_LAZY (*C macro*), 972
HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY (*C macro*), 1348
command line option, 60
HPX_INVOKE_R (*C macro*), 1036
HPX_ITERATOR_SUPPORT_WITH_BOOST_ITERATOR_TRAVERSAL_TAG (*C macro*), 1348
command line option, 60
HPX_MAKE_EXCEPTIONAL_FUTURE (*C macro*), 1043
HPX_ONCE_INIT (*C macro*), 1143
HPX_PERFORMANCE_COUNTER_V1 (*C macro*), 1348
HPX_PLAIN_ACTION (*C macro*), 1212
HPX_PLAIN_ACTION_ID (*C macro*), 1213
HPX_PP_CAT (*C macro*), 1069
HPX_PP_EXPAND (*C macro*), 1070
HPX_PP_NARGS (*C macro*), 1070
HPX_PP_STRINGIZE (*C macro*), 1071
HPX_PP_STRIP_PARENS (*C macro*), 1071
HPX_REGISTER_ACTION (*C macro*), 1209
HPX_REGISTER_ACTION_DECLARATION (*C macro*), 1208
HPX_REGISTER_ACTION_DECLARATION_ (*C macro*), 1209
HPX_REGISTER_ACTION_DECLARATION_1 (*C macro*), 1209
HPX_REGISTER_ACTION_ID (*C macro*), 1209
HPX_REGISTER_BASE_LCO_WITH_VALUE_ (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_ (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_1 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_2 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_3 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_4 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION2 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_ (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_1 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_2 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID2 (*C macro*), 1240
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_ (*C macro*), 1241
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_4 (*C macro*), 1241
HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_5 (*C macro*), 1241
HPX_REGISTER_BINARY_FILTER_FACTORY (*C macro*), 1363
HPX_REGISTER_COMMANDLINE_MODULE (*C macro*), 1306
HPX_REGISTER_COMMANDLINE_MODULE_DYNAMIC (*C macro*), 1306
HPX_REGISTER_COMMANDLINE_OPTIONS (*C macro*), 1077
HPX_REGISTER_COMMANDLINE_OPTIONS_DYNAMIC (*C macro*), 1077
HPX_REGISTER_COMMANDLINE_REGISTRY (*C macro*), 1077
HPX_REGISTER_COMMANDLINE_REGISTRY_DYNAMIC (*C macro*), 1077
HPX_REGISTER_COMPONENT (*C macro*), 1366
HPX_REGISTER_COMPONENT_REGISTRY (*C macro*), 1078
HPX_REGISTER_COMPONENT_REGISTRY_DYNAMIC (*C macro*), 1078
HPX_REGISTER_DERIVED_COMPONENT_FACTORY (*C macro*), 1370
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_ (*C macro*), 1370
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_3 (*C macro*), 1370
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_4 (*C macro*), 1370
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC (*C macro*), 1370
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_ (*C macro*), 1370
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_3 (*C macro*), 1370
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_4 (*C macro*), 1370
HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY (*C macro*), 1366
HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_ (*C macro*), 1366

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_2 (*C macro*, 1366) HPX_SERIALIZATION_WITH_SUPPORTS_ENDIANESS:BOOL command line option, 60

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_3 (*C macro*, 1366) HPX_THROW_EXCEPTION (*C macro*, 987)
HPX_THROWMODE_UNSCOPED_ENUM_DEPRECATED_MSG

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC (*C macro*, 1366) HPX_THROWS_IF (*C macro*, 987)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC (*C macro*, 1366) HPX_TOPOLOGY_WITH_ADDITIONAL_HWLOC_TESTING:BOOL command line option, 60

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC (*C macro*, 1366) HPX_UTIL_REGISTER_FUNCTION (*C macro*, 1033)
HPX_UTIL_REGISTER_FUNCTION_DECLARATION (*C macro*, 1033)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_3 (*C macro*, 1366) HPX_UTIL_REGISTER_UNIQUE_FUNCTION (*C macro*, 1040)

HPX_REGISTER_PLUGIN_BASE_REGISTRY (*C macro*, 1079) HPX_UTIL_REGISTER_UNIQUE_FUNCTION_DECLARATION (*C macro*, 1040)

HPX_REGISTER_PLUGIN_REGISTRY (*C macro*, 1365) HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL command line option, 57

HPX_REGISTER_PLUGIN_REGISTRY_2 (*C macro*, 1365) HPX_WITH_APEX

HPX_REGISTER_PLUGIN_REGISTRY_4 (*C macro*, 1365) command line option, 43

HPX_REGISTER_PLUGIN_REGISTRY_5 (*C macro*, 1365) HPX_WITH_APEX:BOOL

HPX_REGISTER_PLUGIN_REGISTRY_MODULE (*C macro*, 1079) command line option, 58

HPX_REGISTER_PLUGIN_REGISTRY_MODULE_DYNAMIC (*C macro*, 1079) HPX_WITH_ASIO_TAG:STRING command line option, 54

HPX_REGISTER_REGISTRY_MODULE (*C macro*, 1078) HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL command line option, 59

HPX_REGISTER_REGISTRY_MODULE_DYNAMIC (*C macro*, 1078) HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL command line option, 51

HPX_REGISTER_SHUTDOWN_MODULE (*C macro*, 1307) HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH command line option, 51

HPX_REGISTER_SHUTDOWN_MODULE_DYNAMIC (*C macro*, 1307) HPX_WITH_BUILD_BINARY_PACKAGE:BOOL command line option, 51

HPX_REGISTER_STARTUP_MODULE (*C macro*, 1307) HPX_WITH_CHECK_MODULE_DEPENDENCIES:BOOL command line option, 51

HPX_REGISTER_STARTUP_MODULE_DYNAMIC (*C macro*, 1307) HPX_WITH_COMPILE_ONLY_TESTS:BOOL command line option, 54

HPX_REGISTER_STARTUP_SHUTDOWN_FUNCTIONS (*C macro*, 1080) HPX_WITH_COMPILER_WARNINGS:BOOL command line option, 51

HPX_REGISTER_STARTUP_SHUTDOWN_FUNCTIONS_DYNAMIC (*C macro*, 1081) HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL command line option, 51

HPX_REGISTER_STARTUP_SHUTDOWN_MODULE (*C macro*, 1307) HPX_WITH_COMPRESSION_BZIP2:BOOL command line option, 51

HPX_REGISTER_STARTUP_SHUTDOWN_MODULE_DYNAMIC (*C macro*, 1307) HPX_WITH_COMPRESSION_SNAPPY:BOOL command line option, 52

HPX_REGISTER_STARTUP_SHUTDOWN_REGISTRY (*C macro*, 1080) HPX_WITH_COMPRESSION_ZLIB:BOOL command line option, 52

HPX_REGISTER_STARTUP_SHUTDOWN_REGISTRY_DYNAMIC (*C macro*, 1080) HPX_WITH_COROUTINE_COUNTERS:BOOL command line option, 56

HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZED:BOOL command line option, 60

HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS:BOOL command line option, 43

HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION:BOOL command line option, 52

HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL command line option, 43

HPX_WITH_CUDA:BOOL command line option, 43

HPX_WITH_CXX_STANDARD:STRING command line option, 43

command line option, 52
HPX_WITH_DATAPAR:BOOL
 command line option, 52
HPX_WITH_DATAPAR_BACKEND:STRING
 command line option, 52
HPX_WITH_DATAPAR_VC_NO_LIBRARY:BOOL
 command line option, 52
HPX_WITH_DEPRECATED_WARNINGS:BOOL
 command line option, 52
HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL
 command line option, 52
HPX_WITH_DISTRIBUTED_RUNTIME:BOOL
 command line option, 54
HPX_WITH_DOCUMENTATION:BOOL
 command line option, 54
HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS:STRING
 command line option, 54
HPX_WITH_DYNAMIC_HPX_MAIN:BOOL
 command line option, 52
HPX_WITH_EXAMPLES
 command line option, 43
HPX_WITH_EXAMPLES:BOOL
 command line option, 54
HPX_WITH_EXAMPLES_HDF5:BOOL
 command line option, 54
HPX_WITH_EXAMPLES_OPENMP:BOOL
 command line option, 54
HPX_WITH_EXAMPLES_QT4:BOOL
 command line option, 54
HPX_WITH_EXAMPLES_QTHREADS:BOOL
 command line option, 54
HPX_WITH_FAIL_COMPILE_TESTS:BOOL
 command line option, 54
HPX_WITH_FAULT_TOLERANCE:BOOL
 command line option, 52
HPX_WITH_FETCH_ASIO:BOOL
 command line option, 54
HPX_WITH_FETCH_LCI:BOOL
 command line option, 55
HPX_WITH_FULL_RPATH:BOOL
 command line option, 52
HPX_WITH_GCC_VERSION_CHECK:BOOL
 command line option, 52
HPX_WITH_GENERIC_CONTEXT_COROUTINES
 command line option, 43
HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL
 command line option, 52
HPX_WITH_HIDDEN_VISIBILITY:BOOL
 command line option, 52
HPX_WITH_HIP:BOOL
 command line option, 52
HPX_WITH_IO_COUNTERS:BOOL
 command line option, 55
HPX_WITH_IO_POOL:BOOL
 command line option, 56
HPX_WITH_ITTNOTIFY:BOOL
 command line option, 58
HPX_WITH_LCI_TAG:STRING
 command line option, 55
HPX_WITH_LOGGING:BOOL
 command line option, 52
HPX_WITH_MALLOC
 command line option, 43
HPX_WITH_MALLOC:STRING
 command line option, 52
HPX_WITH_MAX_CPU_COUNT
 command line option, 43
HPX_WITH_MAX_CPU_COUNT:STRING
 command line option, 56
HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING
 command line option, 56
HPX_WITH_MODULES_AS_STATIC_LIBRARIES:BOOL
 command line option, 52
HPX_WITH_NETWORKING:BOOL
 command line option, 57
HPX_WITH_NICE_THREADLEVEL:BOOL
 command line option, 53
HPX_WITH_PAPI:BOOL
 command line option, 58
HPX_WITH_PARALLEL_TESTS_BIND_NONE:BOOL
 command line option, 59
HPX_WITH_PARCEL_COALESCING:BOOL
 command line option, 53
HPX_WITH_PARCEL_PROFILING:BOOL
 command line option, 58
HPX_WITH_PARCELPORT_ACTION_COUNTERS:BOOL
 command line option, 57
HPX_WITH_PARCELPORT_COUNTERS:BOOL
 command line option, 58
HPX_WITH_PARCELPORT_LCI:BOOL
 command line option, 58
HPX_WITH_PARCELPORT_LIBFABRIC:BOOL
 command line option, 58
HPX_WITH_PARCELPORT_MPI
 command line option, 43
HPX_WITH_PARCELPORT_MPI:BOOL
 command line option, 58
HPX_WITH_PARCELPORT_TCP
 command line option, 43
HPX_WITH_PARCELPORT_TCP:BOOL
 command line option, 58
HPX_WITH_PKGCONFIG:BOOL
 command line option, 53
HPX_WITH_POWER_COUNTER:BOOL

```

    command line option, 60
HPX_WITH_PRECOMPILED_HEADERS:BOOL
    command line option, 53
HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL
    command line option, 53
HPX_WITH_SANITIZERS:BOOL
    command line option, 59
HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL
    command line option, 56
HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL
    command line option, 56
HPX_WITH_SPINLOCK_POOL_NUM:STRING
    command line option, 56
HPX_WITH_STACKOVERFLOW_DETECTION:BOOL
    command line option, 53
HPX_WITH_STACKTRACES:BOOL
    command line option, 56
HPX_WITH_STACKTRACES_DEMANGLE_SYMBOLS:BOOL
    command line option, 56
HPX_WITH_STACKTRACES_STATIC_SYMBOLS:BOOL
    command line option, 56
HPX_WITH_STATIC_LINKING:BOOL
    command line option, 53
HPX_WITH_TESTS
    command line option, 43
HPX_WITH_TESTS:BOOL
    command line option, 55
HPX_WITH_TESTS_BENCHMARKS:BOOL
    command line option, 55
HPX_WITH_TESTS_DEBUG_LOG:BOOL
    command line option, 59
HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING
    command line option, 59
HPX_WITH_TESTS_EXAMPLES:BOOL
    command line option, 55
HPX_WITH_TESTS_EXTERNAL_BUILD:BOOL
    command line option, 55
HPX_WITH_TESTS_HEADERS:BOOL
    command line option, 55
HPX_WITH_TESTS_MAX_THREADS_PER_LOCALITY:STRING
    command line option, 59
HPX_WITH_TESTS_REGRESSIONS:BOOL
    command line option, 55
HPX_WITH_TESTS_UNIT:BOOL
    command line option, 55
HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING
    command line option, 56
HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL
    command line option, 56
HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL
    command line option, 56
HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL
    command line option, 57
HPX_WITH_THREAD_DEBUG_INFO:BOOL
    command line option, 59
HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL
    command line option, 59
HPX_WITH_THREAD_GUARD_PAGE:BOOL
    command line option, 59
HPX_WITH_THREAD_IDLE_RATES:BOOL
    command line option, 57
HPX_WITH_THREAD_LOCAL_STORAGE:BOOL
    command line option, 57
HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL
    command line option, 57
HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL
    command line option, 57
HPX_WITH_THREAD_STACK_MMAP:BOOL
    command line option, 57
HPX_WITH_THREAD_STEALING_COUNTS:BOOL
    command line option, 57
HPX_WITH_THREAD_TARGET_ADDRESS:BOOL
    command line option, 57
HPX_WITH_TIMER_POOL:BOOL
    command line option, 57
HPX_WITH_TOOLS:BOOL
    command line option, 55
HPX_WITH_UNITY_BUILD:BOOL
    command line option, 53
HPX_WITH_VALGRIND:BOOL
    command line option, 59
HPX_WITH_VERIFY_LOCKS:BOOL
    command line option, 59
HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL
    command line option, 59
HPX_WITH_VIM_YCM:BOOL
    command line option, 53
HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING
    command line option, 53
HWLOC_ROOT:PATH
    command line option, 60

L
lcos (C++ type), 1050, 1245
Lightweight Control Object, 223
Local Control Object, 223
Locality, 223

P
PAPI_ROOT:PATH
    command line option, 61
Parcel, 223
Process, 223

S
SPHINX_ROOT:PATH
    command line option, 1418

```

std (*C++ type*), 953, 1052, 1055, 1165, 1249
std::filesystem (*C++ type*), 1029
std::hash<::hpx::thread::id> (*C++ struct*), 1160
std::hash<::hpx::thread::id>::operator()
 (*C++ function*), 1160
std::hash<::hpx::threads::thread_id_ref>
 (*C++ struct*), 949
std::hash<::hpx::threads::thread_id_ref>::operator()
 (*C++ function*), 949
std::hash<::hpx::threads::thread_id> (*C++
 struct*), 949
std::hash<::hpx::threads::thread_id>::operator()
 (*C++ function*), 949
std::PhonyNameDueToError::operator() (*C++
 function*), 953, 1166
std::swap (*C++ function*), 1052
std::uses_allocator<hpx::distributed::promise<R>,
 Allocator> (*C++ struct*), 1248
std::uses_allocator<hpx::packaged_task<Sig>,
 Allocator> (*C++ struct*), 1051
std::uses_allocator<hpx::promise<R>,
 Allocator> (*C++ struct*), 1052