# HPX 0.9.9

The STE||AR Group

## Table of Contents

Preface ... 3
What's New ... 5
    *HPX* V0.9.9 ... 5
    Previous *HPX* Releases ... 8
Tutorial ... 42
    Getting Started ... 42
    Introduction ... 56
    Examples ... 60
Manual ... 76
    Building *HPX* with CMake ... 76
    Testing *HPX* ... 87
    Launching *HPX* ... 88
    Writing *HPX* applications ... 115
    Global Names ... 118
    Applying Actions ... 118
    Writing Plain Actions ... 126
    Writing Components ... 126
    Using LCOs ... 126
    Error Handling ... 127
    Performance Counters ... 131
    *HPX* Thread Scheduling Policies ... 160
Index ... 163
Reference ... 172
    Header <hpx/error.hpp> ... 172
    Header <hpx/exception.hpp> ... 173
    Header <hpx/hpx_finalize.hpp> ... 199
    Header <hpx/hpx_fwd.hpp> ... 203
    Header <hpx/hpx_init.hpp> ... 228
    Header <hpx/hpx_start.hpp> ... 235
    Header <hpx/lcos/broadcast.hpp> ... 242
    Header <hpx/lcos/fold.hpp> ... 245
    Header <hpx/lcos/reduce.hpp> ... 248
    Header <hpx/performance_counters/manage_counter_type.hpp> ... 249
    Header <hpx/runtime/actions/action_support.hpp> ... 253
    Header <hpx/runtime/actions/component_non_const_action.hpp> ... 256
    Header <hpx/runtime/actions/plain_action.hpp> ... 258
    Header <hpx/runtime/components/copy_component.hpp> ... 261
    Header <hpx/runtime/components/migrate_component.hpp> ... 262
    Header <hpx/runtime/components/new.hpp> ... 263

Header <hpx/runtime/get_ptr.hpp> .... 264
Header <hpx/runtime/naming/unmanaged.hpp> .... 265
Terminology .... 267
People .... 268

The STE||AR Group (**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch) is an international research group with the goal of promoting 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. The main contributors to *HPX* in the STE||AR Group are researchers from Louisiana State University (LSU)'s Center for Computation and Technology (CCT) and the Friedrich-Alexander University Erlangen-Nuremberg (FAU)'s Department of Computer Science 3 - Computer Architecture. For a full list of people working in this group and participating in writing this documentation see People.

This documentation is automatically generated for *HPX* V0.9.9 (from Git commit: 600967a136326a6f21399637de29579e84547373) by the Boost QuickBook and AutoIndex documentation tools. QuickBook and AutoIndex can be found in the collection of Boost Tools.

# Preface

## History

The development of High Performance ParalleX (*HPX*) started back in 2007. At that point, Hartmut Kaiser became interested in the work done by the ParalleX group at the Center for Computation and Technology (CCT), a multi-disciplinary research institute at Louisiana State University (LSU). The ParalleX group was developing a new and experimental execution model for future high performance computing architectures called ParalleX. The first implementations were crude at best, and we had to dismiss those designs entirely. However, over time we learned quite a bit about how to design a parallel distributed runtime system which implements the ideas of ParalleX.

Our goal is to create a high quality, freely available, open source implementation of the ParalleX 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 an stable API which will provide a smooth migration path for developers. The API exposed by *HPX* is modelled after the interfaces defined by the C++11 ISO standard and adheres to the programming guidelines used by the Boost collection of C++ libraries.

From the very beginning, this endeavour has been a group effort. In addition to a handful of interested researchers, there have always been graduate and undergraduate students participating in the discussions, design, and implementation of *HPX*. In 2011 we decided to formalize our collective research efforts by creating the STE||AR group. STE||AR stands for **S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch, which describes the three main focal points we center our work around.

To learn more about STE||AR and ParalleX, see People and Introduction.

## How to use this manual

Some icons are used to mark certain topics indicative of their relevance. These icons precede some text to indicate:

### Table 1. Icons

| Icon | Name | Meaning |
|---|---|---|
| 🔍 | Note | Generally useful information (an aside that doesn't fit in the flow of the text) |
| 💡 | Tip | Suggestion on how to do something (especially something that is not obvious) |
| ℹ️ | Important | Important note on something to take particular notice of |
| ⚠️ | Caution | Take special care with this - it may not be what you expect and may cause bad results |

The following table describes the syntax that will be used to refer to functions and classes throughout the manual:

**Table 2. Syntax for Code References**

| Syntax | Meaning |
|---|---|
| `foo()` | The function foo |
| `foo<>()` | The template function foo (used only for template functions that require explicit parameters) |
| `foo` | The class foo |
| `foo<>` | The class template foo |

## Support

Please feel free to direct questions to *HPX*'s mailing list: hpx-users@stellar.cct.lsu.edu or log onto our IRC channel which can be found at #ste||ar at Freenode.

# What's New

## *HPX* V0.9.9

### General Changes

- 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.

### Bug Fixes (Closed Tickets)

Here is a list of the important tickets we closed for this release.

- #1154 - Fixing iostreams
- #1153 - Standard manipulators (like std::endl) do not work with hpx::ostream
- #1152 - Functions revamp
- #1151 - Supressing cmake 3.0 policy warning for CMP0026
- #1150 - Client Serialization error
- #1149 - Segfault on Stampede
- #1148 - Refactoring mini-ghost
- #1147 - N3960 copy_if and copy_n implemented and tested
- #1146 - Stencil print
- #1145 - N3960 hpx::parallel::copy implemented and tested
- #1144 - OpenMP examples 1d_stencil do not build
- #1143 - 1d_stencil OpenMP examples do not build
- #1142 - Cannot build HPX with gcc 4.6 on OS X
- #1140 - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
- #1139 - hpx/hpx/config/compiler_specific.hpp
- #1137 - Improvements to create binary packages
- #1136 - HPX_GCC_VERSION not defined on all compilers
- #1136 - HPX_GCC_VERSION not defined on all compilers
- #1135 - Avoiding collision between winsock2.h and windows.h
- #1134 - Making sure, that hpx::finalize can be called from any locality
- #1133 - 1d stencil examples

- #1130 - Unique function
- #1128 - Action future args
- #1127 - Executor causes segmentation fault
- #1124 - Adding new API functions: `register_id_with_basename`, `unregister_id_with_basename`, `find_ids_from_basename`; adding test
- #1123 - Reduce nesting of try-catch construct in `encode_parcels`?
- #1122 - Client base fixes
- #1121 - Update `hpxrun.py.in`
- #1120 - HTTS2 tests compile errors on v110 (VS2012)
- #1119 - Remove references to boost::atomic in accumulator example
- #1118 - Only build test thread_pool_executor_1114_test if `HPX_LOCAL_SCHEDULER` is set
- #1117 - local_queue_executor linker error on vc110
- #1116 - Disabled performance counter should give runtime errors, not invalid data
- #1115 - Compile error with Intel C++ 13.1
- #1114 - Default constructed executor is not usable
- #1113 - Fast compilation of logging causes ABI incompatibilities between different `NDEBUG` values
- #1112 - Using thread_pool_executors causes segfault
- #1111 - `hpx::threads::get_thread_data` always returns zero
- #1110 - Remove unnecessary null pointer checks
- #1109 - More tests adjustments
- #1108 - Clarify build rules for "libboost_atomic-mt.so"?
- #1107 - Remove unnecessary null pointer checks
- #1106 - network_storage benchmark imporvements, adding legends to plots and tidying layout
- #1105 - Add more plot outputs and improve instructions doc
- #1104 - Complete quoting for parameters of some CMake commands
- #1103 - Work on test/scripts
- #1102 - Changed minimum requirement of window install to 2012
- #1101 - Changed minimum requirement of window install to 2012
- #1100 - Changed readme to no longer specify using MSVC 2010 compiler
- #1099 - Error returning futures from component actions

- #1098 - Improve storage test
- #1097 - data_actions quickstart example calls missing function decorate_action of data_get_action
- #1096 - MPI parcelport broken with new zero copy optimization
- #1095 - Warning C4005: _WIN32_WINNT: Macro redefinition
- #1094 - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
- #1093 - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS
- #1092 - Rename unique_future<> back to future<>
- #1091 - Inconsistent error message
- #1090 - On windows 8.1 the examples crashed if using more than one os thread
- #1089 - Components should be allowed to have their own executor
- #1088 - Add possibility to select a network interface for the ibverbs parcelport
- #1087 - ibverbs and ipc parcelport uses zero copy optimization
- #1083 - Make shell examples copyable in docs
- #1082 - Implement proper termination detection during shutdown
- #1081 - Implement thread_specific_ptr for hpx::threads
- #1070 - Complete quoting for parameters of some CMake commands
- #1059 - Fix more unused variable warnings
- #973 - Would like option to report hwloc bindings
- #941 - Create a proper user level context switching class for BG/Q
- #935 - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- #879 - Hung test leads to cascading test failure; make tests should support the MPI parcelport
- #865 - future<T> and friends shall work for movable only Ts
- #525 - Extend barrier LCO test to run in distributed
- #509 - Push Boost.Atomic changes upstream
- #461 - Add a compilation sanity test
- #456 - hpx_run_tests.py should log output from tests that timeout
- #454 - Investigate threadmanager performance
- #190 - hpx::cout should be a std::ostream
- #189 - iostreams component should use startup/shutdown functions
- #183 - Use Boost.ICL for correctness in AGAS

- #44 - Implement real futures

# Previous *HPX* Releases

## *HPX* V0.9.8

We have had over 800 commits since the last release and we have closed over 65 tickets (bugs, feature requests, etc.).

With the changes below, *HPX* is once again leading the charge of a whole new era of computation. By intrinsically breaking down and synchronizing the work to be done, *HPX* insures that application developers will no longer have to fret about where a segment of code executes. That allows coders to focus their time and energy to understanding the data dependencies of their algorithms and thereby the core obstacles to an efficient code. Here are some of the advantages of using *HPX*:

- *HPX* is solidly rooted in a sophisticated theoretical execution model -- ParalleX
- *HPX* exposes an API fully conforming to the C++11 and the draft C++14 standards, extended and applied to distributed computing. Everything programmers know about the concurrency primitives of the standard C++ library is still valid in the context of *HPX*.
- It provides a competitive, high performance implementation of modern, future-proof ideas which gives an smooth migration path from todays mainstream techniques
- There is no need for the programmer to worry about lower level parallelization paradigms like threads or message passing; no need to understand pthreads, MPI, OpenMP, or Windows threads, etc.
- There is no need to think about different types of parallelism such as tasks, pipelines, or fork-join, task or data parallelism.
- The same source of your program compiles and runs on Linux, BlueGene/Q, Mac OS X, Windows, and Android.
- The same code runs on shared memory multi-core systems and supercomputers, on handheld devices and Intel® Xeon Phi™ accelerators, or a heterogeneous mix of those.

### General Changes

- A major API breaking change for this release was introduced by implementing `hpx::future` and `hpx::shared_future` fully in conformance with the C++11 Standard. While `hpx::shared_future` is new and will not create any compatibility problems, we revised the interface and implementation of the existing `hpx::future`. For more details please see the mailing list archive. To avoid any incompatibilities for existing code we named the type which implements the `std::future` interface as `hpx::unique_future`. For the next release this will be renamed to `hpx::future`, making it full conforming to C++11 Standard.
- A large part of the code base of *HPX* has been refactored and partially re-implemented. The main changes were related to
  - The threading subsystem: these changes significantly reduce the amount of overheads caused by the schedulers, improve the modularity of the code base, and extend the variety of available scheduling algorithms.
  - The parcel subsystem: these changes improve the performance of the *HPX* networking layer, modularize the structure of the parcelports, and simplify the creation of new parcelports for other underlying networking libraries.
  - The API subsystem: these changes improved the conformance of the API to C++11 Standard, extend and unify the available API functionality, and decrease the overheads created by various elements of the API.
  - The robustness of the component loading subsystem has been improved significantly, allowing to more portably and more reliably register the components needed by an application as startup. This additionally speeds up general application initialization.
- We added new API functionality like `hpx::migrate` and `hpx::copy_component` which are the basic building blocks necessary for implementing higher level abstractions for system-wide load balancing, runtime-adaptive resource management, and object-oriented checkpointing and state-management.

- We removed the use of C++11 move emulation (using Boost.Move), replacing it with C++11 rvalue references. This is the first step towards using more and more native C++11 facilities which we plan to introduce in the future.
- We improved the reference counting scheme used by *HPX* which helps managing distributed objects and memory. This improves the overall stability of *HPX* and further simplifies writing real world applications.
- The minimal Boost version required to use *HPX* is now V1.49.0.
- This release coincides with the first release of HPXPI (V0.1.0), the first implementation of the XPI specification.

## Bug Fixes (Closed Tickets)

Here is a list of the important tickets we closed for this release.

- #1086 - Expose internal boost::shared_array to allow user management of array lifetime
- #1083 - Make shell examples copyable in docs
- #1080 - /threads{locality#*/total}/count/cumulative broken
- #1079 - Build problems on OS X
- #1078 - Improve robustness of component loading
- #1077 - Fix a missing enum definition for 'take' mode
- #1076 - Merge Jb master
- #1075 - Unknown CMake command "add_hpx_pseudo_target"
- #1074 - Implement `apply_continue_callback` and `apply_colocated_callback`
- #1073 - The new `apply_colocated` and `async_colocated` functions lead to automatic registered functions
- #1071 - Remove deferred_packaged_task
- #1069 - serialize_buffer with allocator fails at destruction
- #1068 - Coroutine include and forward declarations missing
- #1067 - Add allocator support to util::serialize_buffer
- #1066 - Allow for MPI_Init being called before HPX launches
- #1065 - AGAS cache isn't used/populated on worker localities
- #1064 - Reorder includes to ensure ws2 includes early
- #1063 - Add `hpx::runtime::suspend` and `hpx::runtime::resume`
- #1062 - Fix `async_continue` to propery handle return types
- #1061 - Implement `async_colocated` and `apply_colocated`
- #1060 - Implement minimal component migration
- #1058 - Remove `HPX_UTIL_TUPLE` from code base
- #1057 - Add performance counters for threading subsystem

- #1055 - Thread allocation uses two memory pools
- #1053 - Work stealing flawed
- #1052 - Fix a number of warnings
- #1049 - Fixes for TLS on OSX and more reliable test running
- #1048 - Fixing after 588 hang
- #1047 - Use port '0' for networking when using one locality
- #1046 - `composable_guard` test is broken when having more than one thread
- #1045 - Security missing headers
- #1044 - Native TLS on FreeBSD via __thread
- #1043 - `async` et.al. compute the wrong result type
- #1042 - `async` et.al. implicitly unwrap reference_wrappers
- #1041 - Remove redundant costly Kleene stars from regex searches
- #1040 - CMake script regex match patterns has unnecessary kleenes
- #1039 - Remove use of Boost.Move and replace with std::move and real rvalue refs
- #1038 - Bump minimal required Boost to 1.49.0
- #1037 - Implicit unwrapping of futures in async broken
- #1036 - Scheduler hangs when user code attempts to "block" OS-threads
- #1035 - Idle-rate counter always reports 100% idle rate
- #1034 - Symbolic name registration causes application hangs
- #1033 - Application options read in from an options file generate an error message
- #1032 - `hpx::id_type` local reference counting is wrong
- #1031 - Negative entry in reference count table
- #1030 - Implement condition_variable
- #1029 - Deadlock in thread scheduling subsystem
- #1028 - HPX-thread cumulative count performance counters report incorrect value
- #1027 - Expose `hpx::thread_interrupted` error code as a separate exception type
- #1026 - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
- #1025 - `future::wait_for/wait_until` do not remove callback
- #1024 - Remove dependence to boost assert and create hpx assert
- #1023 - Segfaults with tcmalloc

- #1022 - prerequisites link in readme is broken
- #1020 - HPX Deadlock on external synchronization
- #1019 - Convert using `BOOST_ASSERT` to `HPX_ASSERT`
- #1018 - compiling bug with gcc 4.8.1
- #1017 - Possible crash in io_pool executor
- #1016 - Crash at startup
- #1014 - Implement Increment/Decrement Merging
- #1013 - Add more logging channels to enable greater control over logging granularity
- #1012 - `--hpx:debug-hpx-log` and `--hpx:debug-agas-log` lead to non-thread safe writes
- #1011 - After installation, running applications from the build/staging directory no longer works
- #1010 - Mergable decrement requests are not being merged
- #1009 - `--hpx:list-symbolic-names` crashes
- #1007 - Components are not properly destroyed
- #1006 - Segfault/hang in set_data
- #1003 - Performance counter naming issue
- #982 - Race condition during startup
- #912 - OS X: component type not found in map
- #663 - Create a buildbot slave based on Clang 3.2/OSX
- #636 - Expose `this_locality::apply<act>(p1, p2);` for local execution
- #197 - Add `--console=address` option for PBS runs
- #175 - Asynchronous AGAS API

## *HPX* V0.9.7

We have had over 1000 commits since the last release and we have closed over 180 tickets (bugs, feature requests, etc.).

### General Changes

- Ported HPX to BlueGene/Q
- Improved HPX support for Xeon/Phi accelerators
- Reimplemented `hpx::bind`, `hpx::tuple`, and `hpx::function` for better performance and better compliance with the C++11 Standard. Added `hpx::mem_fn`.
- Reworked `hpx::when_all` and `hpx::when_any` for better compliance with the ongoing C++ standardization effort, added heterogeneous version for those functions. Added `hpx::when_any_swapped`.
- Added `hpx::copy` as a precursor for a migrate functionality

- Added `hpx::get_ptr` allowing to directly access the memory underlying a given component
- Added the `hpx::lcos::broadcast`, `hpx::lcos::reduce`, and `hpx::lcos::fold` collective operations
- Added `hpx::get_locality_name` allowing to retrieve the name of any of the localities for the application.
- Added support for more flexible thread affinity control from the HPX command line, such as new modes for `--hpx:bind` (`balanced`, `scattered`, `compact`), improved default settings when running multiple localities on the same node.
- Added experimental executors for simpler thread pooling and scheduling. This API may change in the future as it will stay aligned with the ongoing C++ standardization efforts.
- Massively improved the performance of the HPX serialization code. Added partial support for zero copy serialization of array and bitwise-copyable types.
- General performance improvements of the code related to threads and futures.

## Bug Fixes (Closed Tickets)

Here is a list of the important tickets we closed for this release.

- #1005 - Allow to disable array optimizations and zero copy optimizations for each parcelport
- #1004 - Generate new HPX logo image for the docs
- #1002 - If MPI parcelport is not available, running HPX under mpirun should fail
- #1001 - Zero copy serialization raises assert
- #1000 - Can't connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
- #999 - Optimize `hpx::when_n`
- #998 - Fixed const-correctness
- #997 - Making serialize_buffer::data() type save
- #996 - Memory leak in hpx::lcos::promise
- #995 - Race while registering pre-shutdown functions
- #994 - thread_rescheduling regression test does not compile
- #992 - Correct comments and messages
- #991 - setcap cap_sys_rawio=ep for power profiling causes an HPX application to abort
- #989 - Jacobi hangs during execution
- #988 - multiple_init test is failing
- #986 - Can't call a function called "init" from "main" when using `<hpx/hpx_main.hpp>`
- #984 - Reference counting tests are failing
- #983 - thread_suspension_executor test fails
- #980 - Terminating HPX threads don't leave stack in virgin state

- #979 - Static scheduler not in documents
- #978 - Preprocessing limits are broken
- #977 - Make tests.regressions.lcos.future_hang_on_get shorter
- #976 - Wrong library order in pkgconfig
- #975 - Please reopen #963
- #974 - Option pu-offset ignored in fixing_588 branch
- #972 - Cannot use MKL with HPX
- #969 - Non-existent INI files requested on the command line via `--hpx:config` do not cause warnings or errors.
- #968 - Cannot build examples in fixing_588 branch
- #967 - Command line description of `--hpx:queuing` seems wrong
- #966 - `--hpx:print-bind` physical core numbers are wrong
- #965 - Deadlock when building in Release mode
- #963 - Not all worker threads are working
- #962 - Problem with SLURM integration
- #961 - `--hpx:print-bind` outputs incorrect information
- #960 - Fix cut and paste error in documentation of get_thread_priority
- #959 - Change link to boost.atomic in documentation to point to boost.org
- #958 - Undefined reference to intrusive_ptr_release
- #957 - Make tuple standard compliant
- #956 - Segfault with a3382fb
- #955 - `--hpx:nodes` and `--hpx:nodefiles` do not work with foreign nodes
- #954 - Make order of arguments for hpx::async and hpx::broadcast consistent
- #953 - Cannot use MKL with HPX
- #952 - register_[pre_]shutdown_function never throw
- #951 - Assert when number of threads is greater than hardware concurrency
- #948 - `HPX_HAVE_GENERIC_CONTEXT_COROUTINES` conflicts with `HPX_HAVE_FIBER_BASED_COROUTINES`
- #947 - Need MPI_THREAD_MULTIPLE for backward compatibility
- #946 - HPX does not call `MPI_Finalize`
- #945 - Segfault with `hpx::lcos::broadcast`
- #944 - OS X: assertion 'pu_offset_ < hardware_concurrency' failed

- #943 - #include <hpx/hpx_main.hpp> does not work
- #942 - Make the BG/Q work with -O3
- #940 - Use separator when concatenating locality name
- #939 - Refactor MPI parcelport to use `MPI_Wait` instead of multiple `MPI_Test` calls
- #938 - Want to officially access client_base::gid_
- #937 - client_base::gid_ should be private
- #936 - Want doxygen-like source code index
- #935 - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- #933 - Cannot build HPX with Boost 1.54.0
- #932 - Components are destructed too early
- #931 - Make HPX work on BG/Q
- #930 - make git-docs is broken
- #929 - Generating index in docs broken
- #928 - Optimize hpx::util::static_ for C++11 compilers supporting magic statics
- #924 - Make kill_process_tree (in process.py) more robust on Mac OSX
- #923 - Correct BLAS and RNPL cmake tests
- #922 - Cannot link against BLAS
- #921 - Implement `hpx::mem_fn`
- #920 - Output locality with `--hpx:print-bind`
- #919 - Correct grammar; simplify boolean expressions
- #918 - Link to hello_world.cpp is broken
- #917 - adapt cmake file to new boostbook version
- #916 - fix problem building documentation with xsltproc >= 1.1.27
- #915 - Add another TBBMalloc library search path
- #914 - Build problem with Intel compiler on Stampede (TACC)
- #913 - fix error messages in fibonacci examples
- #911 - Update OS X build instructions
- #910 - Want like to specify MPI_ROOT instead of compiler wrapper script
- #909 - Warning about void* arithmetic
- #908 - Buildbot for MIC is broken

- #906 - Can't use `--hpx:bind=balanced` with multiple MPI processes
- #905 - `--hpx:bind` documentation should describe full grammar
- #904 - Add hpx::lcos::fold and hpx::lcos::inverse_fold collective operation
- #903 - Add `hpx::when_any_swapped()`
- #902 - Add `hpx::lcos::reduce` collective operation
- #901 - Web documentation is not searchable
- #900 - Web documentation for trunk has no index
- #898 - Some tests fail with GCC 4.8.1 and MPI parcel port
- #897 - HWLOC causes failures on Mac
- #896 - pu-offset leads to startup error
- #895 - `hpx::get_locality_name` not defined
- #894 - Race condition at shutdown
- #893 - `--hpx:print-bind` switches std::cout to hexadecimal mode
- #892 - `hwloc_topology_load` can be expensive -- don't call multiple times
- #891 - The documentation for `get_locality_name` is wrong
- #890 - `--hpx:print-bind` should not exit
- #889 - `--hpx:debug-hpx-log=FILE` does not work
- #888 - MPI parcelport does not exit cleanly for --hpx:print-bind
- #887 - Choose thread affinities more cleverly
- #886 - Logging documentation is confusing
- #885 - Two threads are slower than one
- #884 - is_callable failing with member pointers in C++11
- #883 - Need help with is_callable_test
- #882 - tests.regressions.lcos.future_hang_on_get does not terminate
- #881 - tests/regressions/block_matrix/matrix.hh won't compile with GCC 4.8.1
- #880 - HPX does not work on OS X
- #878 - `future::unwrap` triggers assertion
- #877 - "make tests" has build errors on Ubuntu 12.10
- #876 - tcmalloc is used by default, even if it is not present
- #875 - global_fixture is defined in a header file

- #874 - Some tests take very long
- #873 - Add block-matrix code as regression test
- #872 - HPX documentation does not say how to run tests with detailed output
- #871 - All tests fail with "make test"
- #870 - Please explicitly disable serialization in classes that don't support it
- #868 - boost_any test failing
- #867 - Reduce the number of copies of `hpx::function` arguments
- #863 - Futures should not require a default constructor
- #862 - value_or_error shall not default construct its result
- #861 - `HPX_UNUSED` macro
- #860 - Add functionality to copy construct a component
- #859 - `hpx::endl` should flush
- #858 - Create `hpx::get_ptr<>` allowing to access component implementation
- #855 - Implement `hpx::INVOKE`
- #854 - `hpx/hpx.hpp` does not include `hpx/include/iostreams.hpp`
- #853 - Feature request: null future
- #852 - Feature request: Locality names
- #851 - `hpx::cout` output does not appear on screen
- #849 - All tests fail on OS X after installing
- #848 - Update OS X build instructions
- #846 - Update hpx_external_example
- #845 - Issues with having both debug and release modules in the same directory
- #844 - Create configuration header
- #843 - Tests should use CTest
- #842 - Remove buffer_pool from MPI parcelport
- #841 - Add possibility to broadcast an index with hpx::lcos::broadcast
- #838 - Simplify `util::tuple`
- #837 - Adopt boost::tuple tests for `util::tuple`
- #836 - Adopt boost::function tests for `util::function`
- #835 - Tuple interface missing pieces

- #833 - Partially preprocessing files not working
- #832 - Native papi counters do not work with wild cards
- #831 - Arithmetics counter fails if only one parameter is given
- #830 - Convert hpx::util::function to use new scheme for serializing its base pointer
- #829 - Consistently use `decay<T>` instead of `remove_const< remove_reference<T>>`
- #828 - Update future implementation to N3721 and N3722
- #827 - Enable MPI parcelport for bootstrapping whenever application was started using mpirun
- #826 - Support command line option `--hpx:print-bind` even if `--hpx::bind` was not used
- #825 - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- #824 - Enable lambda functions to be used with hpx::async/hpx::apply
- #823 - Using a hashing filter
- #822 - Silence unused variable warning
- #821 - Detect if a function object is callable with given arguments
- #820 - Allow wildcards to be used for performance counter names
- #819 - Make the AGAS symbolic name registry distributed
- #818 - Add future::then() overload taking an executor
- #817 - Fixed typo
- #815 - Create an lco that is performing an efficient broadcast of actions
- #814 - Papi counters cannot specify thread#* to get the counts for all threads
- #813 - Scoped unlock
- #811 - simple_central_tuplespace_client run error
- #810 - ostream error when << any objects
- #809 - Optimize parcel serialization
- #808 - HPX applications throw exception when executed from the build directory
- #807 - Create performance counters exposing overall AGAS statistics
- #795 - Create timed make_ready_future
- #794 - Create heterogeneous `when_all`/`when_any`/etc.
- #721 - Make HPX usable for Xeon Phi
- #694 - CMake should complain if you attempt to build an example without its dependencies
- #692 - SLURM support broken

- #683 - python/hpx/process.py imports epoll on all platforms
- #619 - Automate the doc building process
- #600 - GTC performance broken
- #577 - Allow for zero copy serialization/networking
- #551 - Change executable names to have debug postfix in Debug builds
- #544 - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init
- #534 - `hpx::init` should take functions by `std::function` and should accept all forms of hpx_main
- #508 - FindPackage fails to set FOO_LIBRARY_DIR
- #506 - Add cmake support to generate ini files for external applications
- #470 - Changing build-type after configure does not update boost library names
- #453 - Document `hpx_run_tests.py`
- #445 - Significant performance mismatch between MPI and HPX in SMP for allgather example
- #443 - Make docs viewable from build directory
- #421 - Support multiple HPX instances per node in a batch environment like PBS or SLURM
- #316 - Add message size limitation
- #249 - Clean up locking code in big boot barrier
- #136 - Persistent CMake variables need to be marked as cache variables

## *HPX* V0.9.6

We have had over 1200 commits since the last release and we have closed roughly 140 tickets (bugs, feature requests, etc.).

### General Changes

The major new fetures in this release are:

- We further consolidated the API exposed by *HPX*. We aligned our APIs as much as possible with the existing C++11 Standard and related proposals to the C++ standardization committee (such as N3632 and N3634).
- We implemented a first version of a distributed AGAS service which essentially eliminates all explicit AGAS network traffic.
- We created a native ibverbs parcelport allowing to take advantage of the superior latency and bandwidth characteristics of Infiniband networks.
- We successfully ported *HPX* to the Xeon Phi platform.
- Support for the SLURM scheduling system was implemented.
- Major efforts have been dedicated to improving the performance counter framework, numerous new counters were implemented and new APIs were added.
- We added a modular parcel compression system allowing to improve bandwidth utilization (by reducing the overall size of the tranferred data).

- We added a modular parcel coalescing system allowing to combine several parcels into larger messages. This reduces latencies introduced by the communication layer.
- Added an experimental executors API allowing to use different scheduling policies for different parts of the code. This API has been modelled after the Standards proposal N3562. 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).

## 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.

- #806 - make (all) in examples folder does nothing
- #805 - Adding the introduction and fixing DOCBOOK dependencies for Windows use
- #804 - Add stackless (non-suspendable) thread type
- #803 - Create proper serialization support functions for util::tuple
- #800 - Add possibility to disable array optimizations during serialization
- #798 - HPX_LIMIT does not work for local dataflow
- #797 - Create a parcelport which uses MPI
- #796 - Problem with Large Numbers of Threads
- #793 - Changing dataflow test case to hang consistently
- #792 - CMake Error
- #791 - Problems with local::dataflow
- #790 - wait_for() doesn't compile
- #789 - HPX with Intel compiler segfaults
- #788 - Intel compiler support
- #787 - Fixed SFINAEd specializations
- #786 - Memory issues during benchmarking.
- #785 - Create an API allowing to register external threads with HPX
- #784 - util::plugin is throwing an error when a symbol is not found
- #783 - How does hpx:bind work?
- #782 - Added quotes around STRING REPLACE potentially empty arguments

- #781 - Make sure no exceptions propagate into the thread manager
- #780 - Allow arithmetics performance counters to expand its parameters
- #779 - Test case for 778
- #778 - Swapping futures segfaults
- #777 - hpx::lcos::details::when_xxx don't restore completion handlers
- #776 - Compiler chokes on dataflow overload with launch policy
- #775 - Runtime error with local dataflow (copying futures?)
- #774 - Using local dataflow without explicit namespace
- #773 - Local dataflow with unwrap: functor operators need to be const
- #772 - Allow (remote) actions to return a future
- #771 - Setting HPX_LIMIT gives huge boost MPL errors
- #770 - Add launch policy to (local) dataflow
- #769 - Make compile time configuration information available
- #768 - Const correctness problem in local dataflow
- #767 - Add launch policies to async
- #766 - Mark data structures for optimized (array based) serialization
- #765 - Align hpx::any with N3508: Any Library Proposal (Revision 2)
- #764 - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations
- #762 - added a human readable output for the ping pong example
- #761 - Ambiguous typename when constructing derived component
- #760 - Simple components can not be derived
- #759 - make install doesn't give a complete install
- #758 - Stack overflow when using locking_hook<>
- #757 - copy paste error; unsupported function overloading
- #756 - GTCX runtime issue in Gordon
- #755 - Papi counters don't work with reset and evaluate API's
- #753 - cmake bugfix and improved component action docs
- #752 - hpx simple component docs
- #750 - Add hpx::util::any
- #749 - Thread phase counter is not reset

- #748 - Memory performance counter are not registered
- #747 - Create performance counters exposing arithmetic operations
- #745 - apply_callback needs to invoke callback when applied locally
- #744 - CMake fixes
- #743 - Problem Building github version of HPX
- #742 - Remove HPX_STD_BIND
- #741 - assertion 'px != 0' failed: HPX(assertion_failure) for low numbers of OS threads
- #739 - Performance counters do not count to the end of the program or evalution
- #738 - Dedicated AGAS server runs don't work; console ignores -a option.
- #737 - Missing bind overloads
- #736 - Performance counter wildcards do not always work
- #735 - Create native ibverbs parcelport based on rdma operations
- #734 - Threads stolen performance counter total is incorrect
- #733 - Test benchmarks need to be checked and fixed
- #732 - Build fails with Mac, using mac ports clang-3.3 on latest git branch
- #731 - Add global start/stop API for performance counters
- #730 - Performance counter values are apparently incorrect
- #729 - Unhandled switch
- #728 - Serialization of hpx::util::function between two localities causes seg faults
- #727 - Memory counters on Mac OS X
- #725 - Restore original thread priority on resume
- #724 - Performance benchmarks do not depend on main HPX libraries
- #723 - --hpx:nodes=`cat $PBS_NODEFILE` works; --hpx:nodefile=$PBS_NODEFILE does not.
- #722 - Fix binding const member functions as actions
- #719 - Create performance counter exposing compression ratio
- #718 - Add possibility to compress parcel data
- #717 - strip_credit_from_gid has misleading semantics
- #716 - Non-option arguments to programs run using `pbsdsh` must be before `--hpx:nodes`, contrary to directions
- #715 - Re-thrown exceptions should retain the original call site
- #714 - failed assertion in debug mode

- #713 - Add performance counters monitoring connection caches
- #712 - Adjust parcel related performance counters to be connection type specific
- #711 - configuration failure
- #710 - Error "timed out while trying to find room in the connection cache" when trying to start multiple localities on a single computer
- #709 - Add new thread state 'staged' referring to task descriptions
- #708 - Detect/mitigate bad non-system installs of GCC on Redhat systems
- #707 - Many examples do not link with Git HEAD version
- #706 - `hpx::init` removes portions of non-option command line arguments before last = sign
- #705 - Create rolling average and median aggregating performance counters
- #704 - Create performance counter to expose thread queue waiting time
- #703 - Add support to HPX build system to find librcrtool.a and related headers
- #699 - Generalize instrumentation support
- #698 - compilation failure with hwloc absent
- #697 - Performance counter counts should be zero indexed
- #696 - Distributed problem
- #695 - Bad perf counter time printed
- #693 - `--help` doesn't print component specific command line options
- #692 - SLURM support broken
- #691 - exception while executing any application linked with hwloc
- #690 - thread_id_test and thread_launcher_test failing
- #689 - Make the buildbots use hwloc
- #687 - compilation error fix (hwloc_topology)
- #686 - Linker Error for Applications
- #684 - Pinning of service thread fails when number of worker threads equals the number of cores
- #682 - Add performance counters exposing number of stolen threads
- #681 - Add apply_continue for asynchronous chaining of actions
- #679 - Remove obsolete async_callback API functions
- #678 - Add new API for setting/triggering LCOs
- #677 - Add async_continue for true continuation style actions
- #676 - Buildbot for gcc 4.4 broken

- #675 - Partial preprocessing broken
- #674 - HPX segfaults when built with gcc 4.7
- #673 - `use_guard_pages` has inconsistent preprocessor guards
- #672 - External build breaks if library path has spaces
- #671 - release tarballs are tarbombs
- #670 - CMake won't find Boost headers in layout=versioned install
- #669 - Links in docs to source files broken if not installed
- #667 - Not reading ini file properly
- #664 - Adapt new meanings of 'const' and 'mutable'
- #661 - Implement BTL Parcel port
- #655 - Make HPX work with the "decltype" result_of
- #647 - documentation for specifying the number of high priority threads `--hpx:high-priority-threads`
- #643 - Error parsing host file
- #642 - HWLoc issue with TAU
- #639 - Logging potentially suspends a running thread
- #634 - Improve error reporting from parcel layer
- #627 - Add tests for async and apply overloads that accept regular C++ functions
- #626 - hpx/future.hpp header
- #601 - Intel support
- #557 - Remove action codes
- #531 - AGAS request and response classes should use switch statements
- #529 - Investigate the state of hwloc support
- #526 - Make HPX aware of hyper-threading
- #518 - Create facilities allowing to use plain arrays as action arguments
- #473 - hwloc thread binding is broken on CPUs with hyperthreading
- #383 - Change result type detection for hpx::util::bind to use result_of protocol
- #341 - Consolidate route code
- #219 - Only copy arguments into actions once
- #177 - Implement distributed AGAS
- #43 - Support for Darwin (Xcode + Clang)

## *HPX* V0.9.5

We have had over 1000 commits since the last release and we have closed roughly 150 tickets (bugs, feature requests, etc.).

### General Changes

This release is continuing along the lines of code and API consolidation, and overall usability inprovements. We dedicated much attention to performance and we were able to significantly improve the threading and networking subsystems.

We successfully ported *HPX* to the Android platform. *HPX* applications now not only can run on mobile devices, but we support heterogeneous applications running across architecture boundaries. At the Supercomputing Conference 2012 we demonstrated connecting Android tablets to simulations running on a Linux cluster. The Android tablet was used to query performance counters from the Linux simulation and to steer its parameters.

We successfully ported *HPX* to Mac OSX (using the Clang compiler). Thanks to Pyry Jahkola for contributing the corresponding patches. Please see the section How to Install *HPX* on Mac OS for more details.

We made a special effort to make HPX usable in highly concurrent use cases. Many of the HPX API functions which possibly take longer than 100 microseconds to execute now can be invoked asynchronously. We added uniform support for composing futures which simplifies to write asynchronous code. HPX actions (function objects encapsulating possibly concurrent remote function invocations) are now well integrated with all other API facilities such like `hpx::bind`.

All of the API has been aligned as much as possible with established paradigms. HPX now mirrors many of the facilities as defined in the C++11 Standard, such as `hpx::thread`, `hpx::function`, `hpx::future`, etc.

A lot of work has been put into improving the documentation. Many of the API functions are documented now, concepts are explained in detail, and examples are better described than before. The new documentation index enables finding information with lesser effort.

This is the first release of HPX we perform after the move to Github. This step has enabled a wider participation from the community and further encourages us in our decision to release HPX as a true open source library (HPX is licensed under the very liberal Boost Software License).

### 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.

- #666 - Segfault on calling hpx::finalize twice
- #665 - Adding declaration num_of_cores
- #662 - pkgconfig is building wrong
- #660 - Need uninterrupt function
- #659 - Move our logging library into a different namespace
- #658 - Dynamic performance counter types are broken
- #657 - HPX v0.9.5 (RC1) hello_world example segfaulting
- #656 - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- #654 - Integrate the Boost auto_index tool with documentation
- #653 - Make HPX build on OS X + Clang + libc++
- #651 - Add fine-grained control for thread pinning

- #650 - Command line no error message when using -hpx:(anything)
- #645 - Command line aliases don't work in `@file`
- #644 - Terminated threads are not always properly cleaned up
- #640 - `future_data<T>::set_on_completed_` used without locks
- #638 - hpx build with intel compilers fails on linux
- #637 - --copy-dt-needed-entries breaks with gold
- #635 - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- #633 - Re-add examples to final 0.9.5 release
- #632 - Example `thread_aware_timer` is broken
- #631 - FFT application throws error in parcellayer
- #630 - Event synchronization example is broken
- #629 - Waiting on futures hangs
- #628 - Add an `HPX_ALWAYS_ASSERT` macro
- #625 - Port coroutines context switch benchmark
- #621 - New INI section for stack sizes
- #618 - pkg_config support does not work with a HPX debug build
- #617 - hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: 'get_thread_id' was not declared in this scope
- #616 - Change wait_xxx not to use locking
- #615 - Revert visibility 'fix' (fb0b6b8245dad1127b0c25ebafd9386b3945cca9)
- #614 - Fix Dataflow linker error
- #613 - find_here should throw an exception on failure
- #612 - Thread phase doesn't show up in debug mode
- #611 - Make stack guard pages configurable at runtime (initialization time)
- #610 - Co-Locate Components
- #609 - future_overhead
- #608 - `--hpx:list-counter-infos` problem
- #607 - Update Boost.Context based backend for coroutines
- #606 - 1d_wave_equation is not working
- #605 - Any C++ function that has serializable arguments and a serializable return type should be remotable
- #604 - Connecting localities isn't working anymore

- #603 - Do not verify any ini entries read from a file
- #602 - Rename argument_size to type_size/ added implementation to get parcel size
- #599 - Enable locality specific command line options
- #598 - Need an API that accesses the performance counter reporting the system uptime
- #597 - compiling on ranger
- #595 - I need a place to store data in a thread self pointer
- #594 - 32/64 interoperability
- #593 - Warn if logging is disabled at compile time but requested at runtime
- #592 - Add optional argument value to `--hpx:list-counters` and `--hpx:list-counter-infos`
- #591 - Allow for wildcards in performance counter names specified with `--hpx:print-counter`
- #590 - Local promise semantic differences
- #589 - Create API to query performance counter names
- #587 - Add get_num_localities and get_num_threads to AGAS API
- #586 - Adjust local AGAS cache size based on number of localities
- #585 - Error while using counters in HPX
- #584 - counting argument size of actions, initial pass.
- #581 - Remove `RemoteResult` template parameter for `future<>`
- #580 - Add possibility to hook into actions
- #578 - Use angle brackets in HPX error dumps
- #576 - Exception incorrectly thrown when `--help` is used
- #575 - HPX(bad_component_type) with gcc 4.7.2 and boost 1.51
- #574 - `--hpx:connect` command line parameter not working correctly
- #571 - `hpx::wait()` (callback version) should pass the future to the callback function
- #570 - `hpx::wait` should operate on `boost::arrays` and `std::lists`
- #569 - Add a logging sink for Android
- #568 - 2-argument version of `HPX_DEFINE_COMPONENT_ACTION`
- #567 - Connecting to a running HPX application works only once
- #565 - HPX doesn't shutdown properly
- #564 - Partial preprocessing of new component creation interface
- #563 - Add `hpx::start`/`hpx::stop` to avoid blocking main thread

- #562 - All command line arguments swallowed by hpx
- #561 - Boost.Tuple is not move aware
- #558 - `boost::shared_ptr<>` style semantics/syntax for client classes
- #556 - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
- #555 - `BOOST_FORCEINLINE` does not name a type
- #554 - Possible race condition in thread `get_id()`
- #552 - Move enable client_base
- #550 - Add stack size category 'huge'
- #549 - ShenEOS run seg-faults on single or distributed runs
- #545 - `AUTOGLOB` broken for add_hpx_component
- #542 - FindHPX_HDF5 still searches multiple times
- #541 - Quotes around application name in hpx::init
- #539 - Race condititition occuring with new lightweight threads
- #535 - hpx_run_tests.py exits with no error code when tests are missing
- #530 - Thread description(<unknown>) in logs
- #523 - Make thread objects more lightweight
- #521 - `hpx::error_code` is not usable for lightweight error handling
- #520 - Add full user environment to HPX logs
- #519 - Build succeeds, running fails
- #517 - Add a guard page to linux coroutine stacks
- #516 - hpx::thread::detach suspends while holding locks, leads to hang in debug
- #514 - Preprocessed headers for <hpx/apply.hpp> don't compile
- #513 - Buildbot configuration problem
- #512 - Implement action based stack size customization
- #511 - Move action priority into a separate type trait
- #510 - trunk broken
- #507 - no matching function for call to `boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(hpx::threads::linux_topology*)`
- #505 - undefined_symbol regression test currently failing
- #502 - Adding OpenCL and OCLM support to HPX for Windows and Linux
- #501 - find_package(HPX) sets cmake output variables

- #500 - wait_any/wait_all are badly named
- #499 - Add support for disabling pbs support in pbs runs
- #498 - Error during no-cache runs
- #496 - Add partial preprocessing support to cmake
- #495 - Support HPX modules exporting startup/shutdown functions only
- #494 - Allow modules to specify when to run startup/shutdown functions
- #493 - Avoid constructing a string in make_success_code
- #492 - Performance counter creation is no longer synchronized at startup
- #491 - Performance counter creation is no longer synchronized at startup
- #490 - Sheneos on_completed_bulk seg fault in distributed
- #489 - compiling issue with g++44
- #488 - Adding OpenCL and OCLM support to HPX for the MSVC platform
- #487 - FindHPX.cmake problems
- #485 - Change distributing_factory and binpacking_factory to use bulk creation
- #484 - Change `HPX_DONT_USE_PREPROCESSED_FILES` to `HPX_USE_PREPROCESSED_FILES`
- #483 - Memory counter for Windows
- #479 - strange errors appear when requesting performance counters on multiple nodes
- #477 - Create (global) timer for multi-threaded measurements
- #472 - Add partial preprocessing using Wave
- #471 - Segfault stack traces don't show up in release
- #468 - External projects need to link with internal components
- #462 - Startup/shutdown functions are called more than once
- #458 - Consolidate hpx::util::high_resolution_timer and `hpx::util::high_resolution_clock`
- #457 - index out of bounds in `allgather_and_gate` on 4 cores or more
- #448 - Make HPX compile with clang
- #447 - 'make tests' should execute tests on local installation
- #446 - Remove SVN-related code from the codebase
- #444 - race condition in smp
- #441 - Patched Boost.Serialization headers should only be installed if needed
- #439 - Components using `HPX_REGISTER_STARTUP_MODULE` fail to compile with MSVC

- #436 - Verify that no locks are being held while threads are suspended
- #435 - Installing HPX should not clobber existing Boost installation
- #434 - Logging external component failed (Boost 1.50)
- #433 - Runtime crash when building all examples
- #432 - Dataflow hangs on 512 cores/64 nodes
- #430 - Problem with distributing factory
- #424 - File paths referring to XSL-files need to be properly escaped
- #417 - Make dataflow LCOs work out of the box by using partial preprocessing
- #413 - hpx_svnversion.py fails on Windows
- #412 - Make hpx::error_code equivalent to hpx::exception
- #398 - HPX clobbers out-of-tree application specific CMake variables (specifically `CMAKE_BUILD_TYPE`)
- #394 - Remove code generating random port numbers for network
- #378 - ShenEOS scaling issues
- #354 - Create a coroutines wrapper for Boost.Context
- #349 - Commandline option `--localities=N/-lN` should be necessary only on AGAS locality
- #334 - Add auto_index support to cmake based documentation toolchain
- #318 - Network benchmarks
- #317 - Implement network performance counters
- #310 - Duplicate logging entries
- #230 - Add compile time option to disable thread debugging info
- #171 - Add an INI option to turn off deadlock detection independently of logging
- #170 - OSHL internal counters are incorrect
- #103 - Better diagnostics for multiple component/action registerations under the same name
- #48 - Support for Darwin (Xcode + Clang)
- #21 - Build fails with GCC 4.6

## *HPX* V0.9.0

We have had roughly 800 commits since the last release and we have closed approximately 80 tickets (bugs, feature requests, etc.).

### General Changes

- Significant improvements made to the usability of *HPX* in large-scale, distributed environments.
- Renamed `hpx::lcos::packaged_task<>` to `hpx::lcos::packaged_action<>` to reflect the semantic differences to a packaged_task as defined by the C++11 Standard.

- *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::future<>`'s. The member function `hpx::lcos::future::when()` permits futures to be sequentially composed. The helper functions `hpx::wait_all`, `hpx::wait_any`, and `hpx::wait_n` can be used to wait for more than one future at a time.
- *HPX* now exposes `hpx::apply()` and `hpx::async()` as the preferred way of creating (or invoking) any deferred work. These functions are usable with various types of functions, function objects, and actions and provide a uniform way to spawn deferred tasks.
- *HPX* now utilizes `hpx::util::bind` to (partially) bind local functions and function objects, and also actions. Remote bound actions can have placeholders as well.
- *HPX* continuations are now fully polymorphic. The class `hpx::actions::forwarding_continuation` is an example of how the user can write is own types of continuations. It can be used to execute any function as an continuation of a particular action.
- Reworked the action invocation API to be fully conformant to normal functions. Actions can now be invoked using `hpx::apply()`, `hpx::async()`, or using the `operator()` implemented on actions. Actions themselves can now be cheaply instantiated as they do not have any members anymore.
- Reworked the lazy action invocation API. Actions can now be directly bound using `hpx::util::bind()` by passing an action instance as the first argument.
- A minimal *HPX* program now looks like this:

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

int hpx_main()
{
    return hpx::finalize();
}

int main()
{
    return hpx::init();
}
```

  This removes the immediate dependency on the Boost.Program Options library.

**Note**

This minimal version of an *HPX* program does not support any of the default command line arguments (such as --help, or command line options related to PBS). It is suggested to always pass `argc` and `argv` to *HPX* as shown in the example below.

- In order to support those, but still not to depend on Boost.Program Options, the minimal program can be written as:

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

// The arguments for hpx_main can be left off, which very similar to the
// behavior of `main()` as defined by C++.
int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

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

- Added performance counters exposing the number of component instances which are alive on a given locality.
- Added performance counters exposing then number of messages sent and received, the number of parcels sent and received, the number of bytes sent and received, the overall time required to send and receive data, and the overall time required to serialize and deserialize the data.
- Added a new component: `hpx::components::binpacking_factory` which is equivalent to the existing `hpx::components::distributing_factory` component, except that it equalizes the overall population of the components to create. It exposes two factory methods, one based on the number of existing instances of the component type to create, and one based on an arbitrary performance counter which will be queried for all relevant localities.
- Added API functions allowing to access elements of the diagnostic information embedded in the given exception: `hpx::get_locality_id()`, `hpx::get_host_name()`, `hpx::get_process_id()`, `hpx::get_function_name()`, `hpx::get_file_name()`, `hpx::get_line_number()`, `hpx::get_os_thread()`, `hpx::get_thread_id()`, and `hpx::get_thread_description()`.

## Bug Fixes (Closed Tickets)

Here is a list of the important tickets we closed for this release:

- #71 - GIDs that are not serialized via `handle_gid<>` should raise an exception
- #105 - Allow for `hpx::util::functions` to be registered in the AGAS symbolic namespace
- #107 - Nasty threadmanger race condition (reproducible in sheneos_test)
- #108 - Add millisecond resolution to *HPX* logs on Linux
- #110 - Shutdown hang in distributed with release build
- #116 - Don't use TSS for the applier and runtime pointers
- #162 - Move local synchronous execution shortcut from hpx::function to the applier
- #172 - Cache sources in CMake and check if they change manually
- #178 - Add an INI option to turn off ranged-based AGAS caching
- #187 - Support for disabling performance counter deployment
- #202 - Support for sending performance counter data to a specific file
- #218 - boost.coroutines allows different stack sizes, but stack pool is unaware of this

- #231 - Implement movable `boost::bind`
- #232 - Implement movable `boost::function`
- #236 - Allow binding `hpx::util::function` to actions
- #239 - Replace `hpx::function` with `hpx::util::function`
- #240 - Can't specify RemoteResult with lcos::async
- #242 - REGISTER_TEMPLATE support for plain actions
- #243 - `handle_gid<>` support for `hpx::util::function`
- #245 - `*_c_cache code` throws an exception if the queried GID is not in the local cache
- #246 - Undefined references in dataflow/adaptive1d example
- #252 - Problems configuring sheneos with CMake
- #254 - Lifetime of components doesn't end when client goes out of scope
- #259 - CMake does not detect that MSVC10 has lambdas
- #260 - io_service_pool segfault
- #261 - Late parcel executed outside of pxthread
- #263 - Cannot select allocator with CMake
- #264 - Fix allocator select
- #267 - Runtime error for hello_world
- #269 - pthread_affinity_np test fails to compile
- #270 - Compiler noise due to -Wcast-qual
- #275 - Problem with configuration tests/include paths on Gentoo
- #325 - Sheneos is 200-400 times slower than the fortran equivalent
- #331 - `hpx::init()` and hpx_main() should not depend on program_options
- #333 - Add doxygen support to CMake for doc toolchain
- #340 - Performance counters for parcels
- #346 - Component loading error when running hello_world in distributed on MSVC2010
- #362 - Missing initializer error
- #363 - Parcel port serialization error
- #366 - Parcel buffering leads to types incompatible exception
- #368 - Scalable alternative to rand() needed for *HPX*
- #369 - IB over IP is substantially slower than just using standard TCP/IP

- #374 - `hpx::lcos::wait` should work with dataflows and arbitrary classes meeting the future interface
- #375 - Conflicting/ambiguous overloads of `hpx::lcos::wait`
- #376 - Find_HPX.cmake should set CMake variable HPX_FOUND for out of tree builds
- #377 - ShenEOS interpolate bulk and interpolate_one_bulk are broken
- #379 - Add support for distributed runs under SLURM
- #382 - _Unwind_Word not declared in boost.backtrace
- #387 - Doxygen should look only at list of specified files
- #388 - Running `make install` on an out-of-tree application is broken
- #391 - Out-of-tree application segfaults when running in qsub
- #392 - Remove HPX_NO_INSTALL option from cmake build system
- #396 - Pragma related warnings when compiling with older gcc versions
- #399 - Out of tree component build problems
- #400 - Out of source builds on Windows: linker should not receive compiler flags
- #401 - Out of source builds on Windows: components need to be linked with hpx_serialization
- #404 - gfortran fails to link automatically when fortran files are present
- #405 - Inability to specify linking order for external libraries
- #406 - Adapt action limits such that dataflow applications work without additional defines
- #415 - `locality_results` is not a member of `hpx::components::server`
- #425 - Breaking changes to `traits::*result` wrt `std::vector<id_type>`
- #426 - AUTOGLOB needs to be updated to support fortran

## *HPX* V0.8.1

This is a point release including important bug fixes for V0.8.0.

### 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:

- #295 - Don't require install path to be known at compile time.
- #371 - Add hpx iostreams to standard build.
- #384 - Fix compilation with GCC 4.7.
- #390 - Remove keep_factory_alive startup call from ShenEOS; add shutdown call to H5close.

- #393 - 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.

## *HPX* V0.8.0

We have had roughly 1000 commits since the last release and we have closed approximately 70 tickets (bugs, feature requests, etc.).

### General Changes

- Improved PBS support, allowing for arbitrary naming schemes of node-hostnames.
- Finished verification of the reference counting framework.
- Implemented decrement merging logic to optimize the distributed reference counting system.
- Restructured the LCO framework. Renamed `hpx::lcos::eager_future<>` and `hpx::lcos::lazy_future<>` into `hpx::lcos::packaged_task<>` and `hpx::lcos::deferred_packaged_task<>`. Split `hpx::lcos::promise<>` into `hpx::lcos::packaged_task<>` and `hpx::lcos::future<>`. Added 'local' futures (in namespace `hpx::lcos::local`).
- Improved the general performance of local and remote action invocations. This (under certain circumstances) drastically reduces the number of copies created for each of the parameters and return values.
- Reworked the performance counter framework. Performance counters are now created only when needed, which reduces the overall resource requirements. The new framework allows for much more flexible creation and management of performance counters. The new sine example application demonstrates some of the capabilities of the new infrastructure.
- Added a buildbot-based continuous build system which gives instant, automated feedback on each commit to SVN.
- Added more automated tests to verify proper functioning of *HPX*.
- Started to create documentation for *HPX* and its API.
- Added documentation toolchain to the build system.
- Added dataflow LCO.
- Changed default *HPX* command line options to have `hpx:` prefix. For instance, the former option `--threads` is now `--hpx:threads`. This has been done to make ambiguities with possible application specific command line options as unlikely as possible. See the section *HPX* Command Line Options for a full list of available options.
- Added the possibility to define command line aliases. The former short (one-letter) command line options have been predefined as aliases for backwards compatibility. See the section *HPX* Command Line Options for a detailed description of command line option aliasing.
- Network connections are now cached based on the connected host. The number of simultaneous connections to a particular host is now limited. Parcels are buffered and bundled if all connections are in use.
- Added more refined thread affinity control. This is based on the external library Portable Hardware Locality (HWLOC).

- Improved support for Windows builds with CMake.
- Added support for components to register their own command line options.
- Added the possibility to register custom startup/shutdown functions for any component. These functions are guaranteed to be executed by an *HPX* thread.
- Added two new experimental thread schedulers: hierarchy_scheduler and periodic_priority_scheduler. These can be activated by using the command line options `--hpx:queueing=hierarchy` or `--hpx:queueing=periodic`.

### Example Applications

- Graph500 performance benchmark (thanks to Matthew Anderson for contributing this application).
- GTC (Gyrokinetic Toroidal Code): a skeleton for particle in cell type codes.
- Random Memory Access: an example demonstrating random memory accesses in a large array
- ShenEOS example, demonstrating partitioning of large read-only data structures and exposing an interpolation API.
- Sine performance counter demo.
- Accumulator examples demonstrating how to write and use *HPX* components.
- Quickstart examples (like hello_world, fibonacci, quicksort, factorial, etc.) demonstrating simple *HPX* concepts which introduce some of the concepts in *HPX*.
- Load balancing and work stealing demos.

### API Changes

- Moved all local LCOs into a separate namespace `hpx::lcos::local` (for instance, `hpx::lcos::local_mutex` is now `hpx::lcos::local::mutex`).
- Replaced `hpx::actions::function` with `hpx::util::function`. Cleaned up related code.
- Removed `hpx::traits::handle_gid` and moved handling of global reference counts into the corresponding serialization code.
- Changed terminology: `prefix` is now called `locality_id`, renamed the corresponding API functions (such as `hpx::get_prefix`, which is now called `hpx::get_locality_id`).
- Adding `hpx::find_remote_localities()`, and `hpx::get_num_localities()`.
- Changed performance counter naming scheme to make it more bash friendly. The new performance counter naming scheme is now

```
/object{parentname#parentindex/instance#index}/counter#parameters
```

- Added `hpx::get_worker_thread_num` replacing `hpx::threadmanager_base::get_thread_num`.
- Renamed `hpx::get_num_os_threads` to `hpx::get_os_threads_count`.
- Added `hpx::threads::get_thread_count`.
- Restructured the Futures sub-system, renaming types in accordance with the terminology used by the C++11 ISO standard.

### Bug Fixes (Closed Tickets)

Here is a list of the important tickets we closed for this release:

- #31 - Specialize handle_gid<> for examples and tests
- #72 - Fix AGAS reference counting
- #104 - heartbeat throws an exception when decrefing the performance counter it's watching
- #111 - throttle causes an exception on the target application
- #142 - One failed component loading causes an unrelated component to fail
- #165 - Remote exception propagation bug in AGAS reference counting test
- #186 - Test credit exhaustion/splitting (e.g. prepare_gid and symbol NS)
- #188 - Implement remaining AGAS reference counting test cases
- #258 - No type checking of GIDs in stubs classes
- #271 - Seg fault/shared pointer assertion in distributed code
- #281 - CMake options need descriptive text
- #283 - AGAS caching broken (gva_cache needs to be rewritten with ICL)
- #285 - HPX_INSTALL root directory not the same as CMAKE_INSTALL_PREFIX
- #286 - New segfault in dataflow applications
- #289 - Exceptions should only be logged if not handled
- #290 - c++11 tests failure
- #293 - Build target for component libraries
- #296 - Compilation error with Boost V1.49rc1
- #298 - Illegal instructions on termination
- #299 - gravity aborts with multiple threads
- #301 - Build error with Boost trunk
- #303 - Logging assertion failure in distributed runs
- #304 - Exception 'what' strings are lost when exceptions from decode_parcel are reported
- #306 - Performance counter user interface issues
- #307 - Logging exception in distributed runs
- #308 - Logging deadlocks in distributed
- #309 - Reference counting test failures and exceptions
- #311 - Merge AGAS remote_interface with the runtime_support object
- #314 - Object tracking for id_types
- #315 - Remove handle_gid and handle credit splitting in id_type serialization

- #320 - applier::get_locality_id() should return an error value (or throw an exception)
- #321 - Optimization for id_types which are never split should be restored
- #322 - Command line processing ignored with Boost 1.47.0
- #323 - Credit exhaustion causes object to stay alive
- #324 - Duplicate exception messages
- #326 - Integrate Quickbook with CMake
- #329 - --help and --version should still work
- #330 - Create pkg-config files
- #337 - Improve usability of performance counter timestamps
- #338 - Non-std exceptions deriving from std::exceptions in tfunc may be sliced
- #339 - Decrease the number of send_pending_parcels threads
- #343 - Dynamically setting the stack size doesn't work
- #351 - 'make install' does not update documents
- #353 - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs
- #355 - 'make' doesn't do anything after correct configuration
- #356 - Don't use hpx::util::static_ in topology code
- #359 - Infinite recursion in hpx::tuple serialization
- #361 - Add compile time option to disable logging completely
- #364 - Installation seriously broken in r7443

## *HPX* V0.7.0

We have had roughly 1000 commits since the last release and we have closed approximately 120 tickets (bugs, feature requests, etc.).

### General Changes

- Completely removed code related to deprecated AGAS V1, started to work on AGAS V2.1.
- Started to clean up and streamline the exposed APIs (see 'API changes' below for more details).
- Revamped and unified performance counter framework, added a lot of new performance counter instances for monitoring of a diverse set of internal *HPX* parameters (queue lengths, access statistics, etc.).
- Improved general error handling and logging support.
- Fixed several race conditions, improved overall stability, decreased memory footprint, improved overall performance (major optimizations include native TLS support and ranged-based AGAS caching).
- Added support for running *HPX* applications with PBS.
- Many updates to the build system, added support for gcc 4.5.x and 4.6.x, added C++11 support.

- Many updates to default command line options.
- Added many tests, set up buildbot for continuous integration testing.
- Better shutdown handling of distributed applications.

## Example Applications

- quickstart/factorial and quickstart/fibonacci, future-recursive parallel algorithms.
- quickstart/hello_world, distributed hello world example.
- quickstart/rma, simple remote memory access example
- quickstart/quicksort, parallel quicksort implementation.
- gtc, gyrokinetic torodial code.
- bfs, breadth-first-search, example code for a graph application.
- sheneos, partitioning of large data sets.
- accumulator, simple component example.
- 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:

- #28 - Integrate Windows/Linux CMake code for *HPX* core
- #32 - hpx::cout() should be hpx::cout

- #33 - AGAS V2 legacy client does not properly handle error_code
- #60 - AGAS: allow for registerid to optionally take ownership of the gid
- #62 - adaptive1d compilation failure in Fusion
- #64 - Parcel subsystem doesn't resolve domain names
- #83 - No error handling if no console is available
- #84 - No error handling if a hosted locality is treated as the bootstrap server
- #90 - Add general commandline option -N
- #91 - Add possibility to read command line arguments from file
- #92 - Always log exceptions/errors to the log file
- #93 - Log the command line/program name
- #95 - Support for distributed launches
- #97 - Attempt to create a bad component type in AMR examples
- #100 - factorial and factorial_get examples trigger AGAS component type assertions
- #101 - Segfault when hpx::process::here() is called in fibonacci2
- #102 - unknown_component_address in int_object_semaphore_client
- #114 - marduk raises assertion with default parameters
- #115 - Logging messages for SMP runs (on the console) shouldn't be buffered
- #119 - marduk linking strategy breaks other applications
- #121 - pbsdsh problem
- #123 - marduk, dataflow and adaptive1d fail to build
- #124 - Lower default preprocessing arity
- #125 - Move hpx::detail::diagnostic_information out of the detail namespace
- #126 - Test definitions for AGAS reference counting
- #128 - Add averaging performance counter
- #129 - Error with endian.hpp while building adaptive1d
- #130 - Bad initialization of performance counters
- #131 - Add global startup/shutdown functions to component modules
- #132 - Avoid using auto_ptr
- #133 - On Windows hpx.dll doesn't get installed
- #134 - HPX_LIBRARY does not reflect real library name (on Windows)

- #135 - Add detection of unique_ptr to build system
- #137 - Add command line option allowing to repeatedly evaluate performance counters
- #139 - Logging is broken
- #140 - CMake problem on windows
- #141 - Move all non-component libraries into $PREFIX/lib/hpx
- #143 - adaptive1d throws an exception with the default command line options
- #146 - Early exception handling is broken
- #147 - Sheneos doesn't link on Linux
- #149 - sheneos_test hangs
- #154 - Compilation fails for r5661
- #155 - Sine performance counters example chokes on chrono headers
- #156 - Add build type to --version
- #157 - Extend AGAS caching to store gid ranges
- #158 - r5691 doesn't compile
- #160 - Re-add AGAS function for resolving a locality to its prefix
- #168 - Managed components should be able to access their own GID
- #169 - Rewrite AGAS future pool
- #179 - Complete switch to request class for AGAS server interface
- #182 - Sine performance counter is loaded by other examples
- #185 - Write tests for symbol namespace reference counting
- #191 - Assignment of read-only variable in point_geometry
- #200 - Seg faults when querying performance counters
- #204 - --ifnames and suffix stripping needs to be more generic
- #205 - --list-* and --print-counter-* options do not work together and produce no warning
- #207 - Implement decrement entry merging
- #208 - Replace the spinlocks in AGAS with hpx::lcos::local_mutexes
- #210 - Add an --ifprefix option
- #214 - Performance test for PX-thread creation
- #216 - VS2010 compilation
- #222 - r6045 context_linux_x86.hpp

- #223 - fibonacci hangs when changing the state of an active thread
- #225 - Active threads end up in the FEB wait queue
- #226 - VS Build Error for Accumulator Client
- #228 - Move all traits into namespace hpx::traits
- #229 - Invalid initialization of reference in thread_init_data
- #235 - Invalid GID in iostreams
- #238 - Demangle type names for the default implementation of get_action_name
- #241 - C++11 support breaks GCC 4.5
- #247 - Reference to temporary with GCC 4.4
- #248 - Seg fault at shutdown with GCC 4.4
- #253 - Default component action registration kills compiler
- #272 - G++ unrecognized command line option
- #273 - quicksort example doesn't compile
- #277 - Invalid CMake logic for Windows

# Tutorial

## Getting Started

### Welcome

Welcome to the *HPX* runtime system libraries! By the time you've completed this tutorial, you'll be at least somewhat comfortable with *HPX* and how to go about using it.

### What's Here

This document is designed to be an extremely gentle introduction, so we included a fair amount of material that may already be very familiar to you. To keep things simple, we also left out some information intermediate and advanced users will probably want. At the end of this document, we'll refer you to resources that can help you pursue these topics further.

## Prerequisites

### Supported Platforms

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

**Table 3. Supported Platforms for *HPX***

| Name | Recommended Version | Minimum Version | Architectures |
|---|---|---|---|
| Linux | 3.2 | 2.6 | x86-32, x86-64, k1om |
| BlueGeneQ | V1R2M0 | V1R2M0 | PowerPC A2 |
| Windows | 7, Server 2008 R2 | Any NT system | x86-64 |

### Software and Libraries

In the simplest case, *HPX* depends on one set of libraries: Boost. So, before you read further, please make sure you have a recent version of Boost installed on your target machine. *HPX* currently requires at least Boost V1.47.0 to work properly. It may build and run with older versions, but we do not test *HPX* with those versions, so please be warned.

Installing the Boost libraries is described in detail in Boost's own Getting Started document. It is often possible to download the Boost libraries using the package manager of your distribution. Please refer to the corresponding documentation for your system for more information.

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

In addition, we urge every user to have a recent version of hwloc installed on the target system in order to have proper support for thread pinning and NUMA awareness.

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

### Table 4. Software Prerequisites for *HPX* on Linux systems

| Name | Recommended Version | Minimum Version | |
|---|---|---|---|
| **Compilers** | | | |
| GNU Compiler Collection (g++) | 4.6.3 or newer | 4.4.5 | |
| Intel Composer XE Suites | 2013 | 2013 | |
| __clang__ | 3.3 or newer | 3.3 | |
| **Build System** | | | |
| CMake | 2.8.4 | 2.8.4 | |
| **Required Libraries** | | | |
| Boost C++ Libraries | 1.51.0 or newer | 1.49.0 | See below for an important limitation when using Boost V1.54.0. |
| Portable Hardware Locality (HWLOC) | 1.8 | 1.2 | Used for OS-thread pinning and NUMA awareness. |

**Important**

Because of a problem in Boost V1.54.0 this version can't be used for compiling *HPX* if you use gcc V4.6.x. Please use either an earlier or a later version of Boost with this compiler.

### Table 5. Software Prerequisites for *HPX* on Windows systems

| Name | Recommended Version | Minimum Version | |
|---|---|---|---|
| **Compilers** | | | |
| Visual C++ (x64) | 2013 | 2012 | |
| **Build System** | | | |
| CMake | 2.8.4 | 2.8.4 | |
| **Required Libraries** | | | |
| Boost | 1.51.0 or newer | 1.49.0 | |
| Portable Hardware Locality (HWLOC) | 1.8 | 1.5 | Used for OS-thread pinning and NUMA awareness. |

### Note

You need to build the following Boost libraries for *HPX*: Boost.DateTime, Boost.Filesystem, Boost.ProgramOptions, Boost.Regex, Boost.Serialization, Boost.System, Boost.Thread, Boost.Chrono (starting Boost 1.49.0), and Boost.Atomic (starting Boost 1.53.0). It is important to note that MSVC2013 is unable to build Boost 1.55.0, a different boost version or compiler will be required.

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.

### Table 6. Highly Recommended Optional Software Prerequisites for *HPX* on Linux systems

| Name | Recommended Version | Minimum Version | Notes |
|---|---|---|---|
| google-perftools | 1.7.1 | 1.7.1 | Used as a replacement for the system allocator, and for allocation diagnostics. |
| libunwind | 0.99 | 0.97 | Dependency of google-perftools on x86-64, used for stack unwinding. |

### Table 7. Optional Software Prerequisites for *HPX* on Linux systems

| Name | Recommended Version | Minimum Version | Notes |
|---|---|---|---|
| Networking and Cryptography library (NaCl) | | | Used for accessing hardware performance data. |
| jemalloc | 2.1.2 | 2.1.0 | Used as a replacement for the system allocator. |
| Hierarchical Data Format V5 (HDF5) | 1.8.7 | 1.6.7 | Used for data I/O in some example applications. See important note below. |

### Table 8. Optional Software Prerequisites for *HPX* on Windows systems

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

### Important

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

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

## Installing Boost Libraries

**Important**

Because of a problem in Boost V1.54.0 this version can't be used for compiling *HPX* if you use gcc V4.6.x. Please use either an earlier or a later version of Boost with this compiler.

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

Building and installing the Boost binaries is simple, regardless what platform you are on:

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

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

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

**Important**

On Windows, depending on the installed versions of Visual Studio, you might also want to pass the correct toolset to the b2 command depending on which version of the IDE you want to use. In addition, passing address-model=64 is always safe

## How to Install *HPX* on Unix Variants

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

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

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

```
cmake --DBOOST_ROOT=/root/of/boost/installation \
      --DHWLOC_ROOT=/root/of/hwloc/installation
      [other CMake variable definitions] \
      -/path/to/source/tree
```

for instance:

```
cmake --DBOOST_ROOT=~/packages/boost --DHWLOC=/packages/hwloc --DCMAKE_INSTALL_PREFIX=~/packages/
hpx ~/downloads/hpx_0.9.6
```

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

```
gmake --j4
```

**Caution**

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

**Note**

Many Linux distributions use `make` as an alias for `gmake`

- To complete the build and install *HPX*:

```
gmake install
```

**Important**

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

```
gmake tests
```

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

```
gmake examples
gmake install
```

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

## How to Install *HPX* on BlueGene/Q

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

- Check if BGClang is available on your installation. If not obtain and install a copy from the `BGClang trac page <https://trac.alcf.anl.gov/projects/llvm-bgq>_`
- Build (and install) a recent version of hwloc Downloads With the following commands:

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

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

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

You can then use this as your build command:

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

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

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

- Generate the *HPX* buildfiles using cmake:

```
cmake --DHPX_PLATFORM=BlueGeneQ \
        --DCMAKE_CXX_COMPILER=bgclang++11 \
        --DMPI_CXX_COMPILER=mpiclang++11 \
        --DHWLOC_ROOT=/path/to/hwloc/installation \
        --DBOOST_ROOT=/path/to/boost \
        --DHPX_MALLOC=system \
        -/path/to/hpx
```

- To complete the build and install *HPX*:

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

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

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

```
to build and install the examples.
```

## How to Install *HPX* on OS X (Mac)

This section describes how to build*HPX* for OS X (Mac) (with recent versions of Clang, libc++, and Boost).

The standard system compiler on OS X is too old to build HPX. You will have to install a newer compiler manually, either Clang or GCC. Below we describe two possibilities:

### Install a recent version of LLVM and Clang

```
In order to build HPX you will need a fairly recent version of Clang
(at least version 3.2 of Clang and LLVM). For more instructions please see
[link http://clang.llvm.org/get_started.html here].

If you're using Homebrew, `brew install llvm ---with-clang` will do the trick
and install 3.2 into `/usr/local/bin`.

In the guide below, I assume you're either using Clang 3.2 as installed by
Homebrew into -/usr/local/bin, or that the following symlinks are created and
visible in your $PATH:
```

```
ln --s -/path/to/build-llvm/Release/bin/clang++ clang++-3.3
ln --s -/path/to/build-llvm/Release/bin/clang clang-3.3
```

(Replace `/path/to` here with an absolute path of your liking.)

### Visit http://libcxx.llvm.org/ to get the latest version of the "libc++" C++ standard library

```
You need to use the trunk version; what's currently bundled with XCode or
OS X aren't quite there yet. You can follow the steps in
http://libcxx.llvm.org/ if you choose, but here's briefly how it can be built:
```

```
cd -/path/to
git clone http://llvm.org/git/libcxx.git
cd libcxx/lib
CXX=clang++-3.3 CC=clang-3.3 TRIPLE=-apple- -./buildit
```

or alternatively:

```
CXX=/usr/local/bin/clang++ CC=/usr/local/bin/clang TRIPLE=-apple- \
    -./buildit
```

The library is then found in `/path/to/libcxx/include` and `/path/to/libcxx/lib`, respectively

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

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

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

# Clang 3.2
using clang
    -: 3.2
    -: -"/usr/local/bin/clang++"
    -: <cxxflags>"-std=c++11 --stdlib=libc++ --fcolor-diagnostics --isystem -/path/to/libcxx/
include"
```

```
      <linkflags>"-stdlib=libc++ --L/path/to/libcxx/lib"
    -;

# Clang trunk ("3.3" for convenience)
using clang
    -: 3.3
    -: -"clang++-3.3"
    -: <cxxflags>"-std=c++11 --stdlib=libc++ --fcolor-diagnostics --isystem -/path/to/libcxx/
include"
      <linkflags>"-stdlib=libc++ --L/path/to/libcxx/lib"
    -;
```

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

```
You can then use as build command either:
```

```
b2 ---build-dir=/tmp/build-boost ---layout=versioned toolset=clang-3.2 install --j4
```

or

```
b2 ---build-dir=/tmp/build-boost ---layout=versioned toolset=clang-3.3 install --j4
```

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

### Build HPX, finally

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

To build with Clang 3.2, execute:

```
cmake -../hpx \
    --DCMAKE_CXX_COMPILER=/usr/local/bin/clang++ \
    --DCMAKE_C_COMPILER=/usr/local/bin/clang-3.3 \
    --DBOOST_INCLUDE_DIR=/usr/local/include/boost-1_53 \
    --DBOOST_LIBRARY_DIR=/usr/local/lib \
    --DBOOST_SUFFIX=-clang-darwin32-mt-1_53 \
    --DCMAKE_CXX_FLAGS="-isystem -/path/to/libcxx/include" \
    --DLINK_FLAGS="-L -/path/to/libcxx/lib"
make
```

To build with Clang 3.3 (trunk), execute:

```
cmake -../hpx \
    --DCMAKE_CXX_COMPILER=clang++-3.3 \
    --DCMAKE_C_COMPILER=clang-3.3 \
    --DBOOST_INCLUDE_DIR=/usr/local/include/boost-1_53 \
    --DBOOST_LIBRARY_DIR=/usr/local/lib \
    --DBOOST_SUFFIX=-clang-darwin33-mt-1_53 \
    --DCMAKE_CXX_FLAGS="-isystem -/path/to/libcxx/include" \
    --DLINK_FLAGS="-L -/path/to/libcxx/lib"
```

```
make
```

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

### Alternative Installation method of HPX on OS X (Mac)

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

1. Install MacPorts
2. Install CMake, gcc 4.8, and hwloc:

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

   You may also want:

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

3. Make this version of gcc your default compiler:

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

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

   ```
   wget http://sourceforge.net/projects/boost/files/boost/1.54.0/boost_1_54_0.tar.bz2
   tar xjf boost_1_54_0.tar.bz2
   pushd boost_1_54_0
   export BOOST_ROOT=$HOME/boost_1_54_0
   ./bootstrap.sh ---prefix=$BOOST_DIR
   ./b2 --j8
   ./b2 --j8 install
   export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:$BOOST_ROOT/lib
   popd
   ```

5. Build HPX:

   ```
   git clone https://github.com/STEllAR-GROUP/hpx.git
   mkdir hpx-build
   pushd hpx-build
   export HPX_ROOT=$HOME/hpx
   cmake --DCMAKE_C_COMPILER=gcc \
       --DCMAKE_CXX_COMPILER=g++ \
       --DCMAKE_FORTRAN_COMPILER=gfortran \
       --DCMAKE_C_FLAGS="-Wno-unused-local-typedefs" \
       --DCMAKE_CXX_FLAGS="-Wno-unused-local-typedefs" \
       --DBOOST_ROOT=$BOOST_ROOT \
       --DHWLOC_ROOT=/opt/local \
       --DCMAKE_INSTALL_PREFIX=$HOME/hpx \
            $(pwd)/../hpx
   ```

```
make --j8
make --j8 install
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:$HPX_ROOT/lib/hpx
popd
```

6. Note that you need to set `BOOST_ROOT`, `HPX_ROOT`, and `DYLD_LIBRARY_PATH` (for both `BOOST_ROOT` and `HPX_ROOT`) every time you configure, build, or run an HPX application.

7. If you want to use HPX with MPI, you need to enable the MPI parcelport, and also specify the location of the MPI wrapper scripts. This can be done e.g. with the following command:

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

## How to Install *HPX* on Windows

### Installation of Required Prerequisites

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

### Installation of the *HPX* Library

- Create a build folder. *HPX* requires an out-of-tree-build. This means that you will be unable to run CMake in the *HPX* source folder.
- Open up the CMake GUI. In the input box labelled "Where is the source code:", enter the full path to the source folder. The source directory is one where the sources were checked out. CMakeLists.txt files in the source directory as well as the subdirectories describe the build to CMake. In addition to this, there are CMake scripts (usually ending in .cmake) stored in a special CMake directory. CMake does not alter any file in the source directory and doesn't add new ones either. In the input box labelled "Where to build the binaries:", enter the full path to the build folder you created before. The build directory is one where all compiler outputs are stored, which includes object files and final executables.
- Add CMake variable definitions (if any) by clicking the "Add Entry" button. There are two required variables you need to define: `BOOST_ROOT` and `HWLOC_ROOT`. These (PATH) variables need to be set to point to the root folder of your Boost and Portable Hardware Locality (HWLOC) installations. It is recommended to set the variable `CMAKE_INSTALL_PREFIX` as well. This determines where the HPX libraries will be built and installed. If this (PATH) variable is set, it has to refer to the directory where the built *HPX* files should be installed to.
- Press the "Configure" button. A window will pop up asking you which compilers to use. Select the Visual Studio 10 (64Bit) compiler (it usually is the default if available). The Visual Studio 2012 (64Bit) and Visual Studio 2013 (64Bit) compilers are supported as well.

Note that while it is possible to build HPX for x86, we don't recommend doing so as 32 bit runs are severely restricted by a 32 bit Windows system limitation affecting the number of HPX threads you can create.

- Make sure that that the `CMAKE_BUILD_TYPE` (under the `CMAKE` tree) is set to `Debug` or `Release`, according to what you plan to do (alternatively, this variable can be set to `RelWithDebInfo` or `MinSizeRel`). The text area at the bottom of CMake GUI displays the output of CMake. CMake prompts an error message in a separate dialog box if a required dependency is not found.
- Press "Configure" again. Repeat this step until the "Generate" button becomes clickable (and until no variable definitions are marked red anymore).
- Press "Generate".
- Open up the build folder, and double-click hpx.sln.
- Build the INSTALL target.

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

## How to Install *HPX* on the Xeon Phi

### Installation of the Boost Libraries

- Download Boost Downloads for Linux and unpack the retreived tarball.
- Adapt your ~/user-config.jam to contain the following lines:

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

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

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

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

You should now have all the required boost libraries.

### Installation of the hwloc Library

- Download hwloc Downloads, unpack the retreived tarball and change to the newly created directory
- Run the configure-make-install procedure as follows:

```
CC=icc CFLAGS=-mmic CXX=icpc CXXFLAGS=-mmic LDFLAGS=-mmic -./configure ---host=k1om ---prefix=
$HWLOC_ROOT
```

```
make
make install
```

You now have a working hwloc installation in $HWLOC_ROOT.

### Building *HPX*

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

```
cmake                             \
    --DCMAKE_CXX_COMPILER=icpc \
    --DCMAKE_C_COMPILER=icc    \
    --DHPX_NATIVE_MIC=On       \
    --DBOOST_ROOT=$BOOST_ROOT  \
    --DHWLOC_ROOT=$HWLOC_ROOT  \
    -/path/to/hpx
```

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

## How to Use *HPX* Applications with PBS

Most *HPX* applications are executed on parallel computers. These platforms typically provide integrated job management services that facilitate the allocation of computing resources for each parallel program. *HPX* includes out of the box support for one of the most common job management systems, the Portable Batch System (PBS).

All PBS jobs require a script to specify the resource requirements and other parameters associated with a parallel job. The PBS script is basically a shell script with PBS directives placed within commented sections at the beginning of the file. The remaining (not commented-out) portions of the file executes just like any other regular shell script. While the description of all available PBS options is outside the scope of this tutorial (the interested reader may refer to in-depth documentation for more information), below is a minimal example to illustrate the approach. As a test application we will use the multithreaded `hello_world` program, explained in the section Hello World Example.

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

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

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

**Caution**

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

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

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

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

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

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

## Important

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

Another choice is for the pbsdsh command in your main job script to invoke your program via a shell, like `sh` or `bash`, so that it gives an initialized environment for each instance. We create a small script `runme.sh` which is used to invoke the program:

```
#!/bin/bash
# Small script which invokes the program based on what was passed on its
# command line.
#
# This script is executed by the bash shell which will initialize all
# environment variables as usual.
$@
```

Now, we invoke this script using the pbsdsh tool:

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

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

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

All that remains now is submitting the job to the queuing system. Assuming that the contents of the PBS script were saved in file `pbs_hello_world.sh` in the current directory, this is accomplished by typing:

```
qsub -./pbs_hello_world_pbs.sh
```

If the job is accepted, qsub will print out the assigned job ID, which may look like:

```
$ 42.supercomputer.some.university.edu
```

To check the status of your job, issue the following command:

```
qstat 42.supercomputer.some.university.edu
```

and look for a single-letter job status symbol. The common cases include:

- **Q** - signifies that the job is queued and awaiting its turn to be executed.
- **R** - indicates that the job is currently running.
- **C** - means that the job has completed.

The example qstat output below shows a job waiting for execution resources to become available:

```
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
42.supercomputer          -...ello_world.sh joe_user               0 Q batch
```

After the job completes, PBS will place two files, `pbs_hello_world.sh.o42` and `pbs_hello_world.sh.e42`, in the directory where the job was submitted. The first contains the standard output and the second contains the standard error from all the nodes on which the application executed. In our example, the error output file should be empty and standard output file should contain something similar to:

```
hello world from OS-thread 3 on locality 0
hello world from OS-thread 2 on locality 0
hello world from OS-thread 1 on locality 1
hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 1
hello world from OS-thread 2 on locality 1
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 1
```

Congratulations! You have just run your first distributed *HPX* application!

## How to Use *HPX* Applications with SLURM

Just like PBS (described in section Using PBS), SLURM is a job management system which is widely used on large supercomputing systems. Any *HPX* application can easily be run using SLURM. This section describes how this can be done.

The easiest way to run an *HPX* application using SLURM is to utilize the command line tool srun which interacts with the SLURM batch scheduling system.

```
srun -p <partition> -N <number-of-nodes> hpx-application <application-arguments>
```

Here, `<partition>` is one of the node partitions existing on the target machine (consult the machines documentation to get a list of existing partitions) and `<number-of-nodes>` is the number of compute nodes you want to use. By default, the HPX application is started with one locality per node and uses all available cores on a node. You can change the number of localities started per node (for example to account for NUMA effects) by specifying the `-n` option of srun. The number of cores per locality can be set by `-c`. The `<application-arguments>` are any application specific arguments which need to passed on to the application.

**Note**

There is no need to use any of the *HPX* command line options related to the number of localities, number of threads, or related to networking ports. All of this information is automatically extracted from the SLURM environment by the *HPX* startup code.

**Important**

The srun documentation explicitly states: "If `-c` is specified without `-n`, as many tasks will be allocated per node as possible while satisfying the `-c` restriction. For instance on a cluster with 8 CPUs per node, a job request for 4 nodes and 3 CPUs per task may be allocated 3 or 6 CPUs per node (1 or 2 tasks per node) depending upon resource consumption by other jobs." For this reason, we suggest to always specify `-n <number-of-instances>`, even if `<number-of-instances>` is equal to one (`1`).

### Interactive Shells

To get an interactive development shell on one of the nodes you can issue the following command:

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

After the shell has been opened, you can run your HPX application. By default, it uses all available cores. Note that if you requested one node, you don't need to do `srun` again. However, if you requested more than one nodes, and want to run your distributed application, you can use `srun` again to start up the distributed HPX application. It will use the resources that have been requested for the interactive shell.

### Scheduling Batch Jobs

The above mentioned method of running *HPX* applications is fine for development purposes. The disadvantage that comes with `srun` is that it only returns once the application is finished. This might not be appropriate for longer running applications (for example benchmarks or larger scale simulations). In order to cope with that limitation you can use the sbatch command.

The `sbatch` command expects a script that it can run once the requested resources are available. In order to request resources you need to add `#SBATCH` comments in your script or provide the necessary parameters to `sbatch` directly. The parameters are the same as with `srun`. The commands you need to execute are the same you would need to start your application as if you were in an interactive shell.

# Introduction

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.

## 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.

### What is *HPX*

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

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

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

*HPX* is designed by the STE||AR Group (**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch) at Louisiana State University (LSU)'s Center for Computation and Technology (CCT) to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STE||AR 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||AR Group, see People.

## What makes our Systems Slow?

Estimates say that we currently run our computers at way below 100% efficiency. The theoretical peak performance (usually measured in FLOPS - floating point operations per second) is much higher than any practical peak performance reached by any application. This is particularly true for highly parallel hardware. The more hardware parallelism we provide to an application, the better the application must scale in order to efficiently use all the resources of the machine. Roughly speaking, we distinguish two forms of scalability: strong scaling (see Amdahl's Law) and weak scaling (see Gustafson's Law). Strong scaling is defined as how the solution time varies with the number of processors for a fixed **total** problem size. It gives an estimate of how much faster can we solve a particular problem by throwing more resources at it. Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size **per processor**. In other words, it defines how much more data can we process by using more hardware resources.

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

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

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

## Technology Demands New Response

Today's computer systems are designed based on the initial ideas of John von Neumann, as published back in 1945, and later extended by the Harvard architecture. These ideas form the foundation, the execution model of computer systems we use currently. But apparently a new response is required in the light of the demands created by today's technology.

So, what are the overarching objectives for designing systems allowing for applications to scale as they should? In our opinion, the main objectives are:

- *Performance*: as mentioned, scalable and efficiency are the main criteria people are interested in
- *Fault tolerance*: the low expected mean time between failures (MTBF) of future systems requires to embrace faults, not trying to avoid them
- *Power*: minimizing energy consumption is a must as it is one of the major cost factors today, even more so in the future
- *Generality*: any system should be usable for a broad set of use cases
- *Programmability*: for me as a programmer this is a very important objective, ensuring long term platform stability and portability

What needs to be done to meet those objectives, to make applications scale better on tomorrow's architectures? Well, the answer is almost obvious: we need to devise a new execution model - a set of governing principles for the holistic design of future systems - targeted at minimizing the effect of the outlined **SLOW** factors. Everything we create for future systems, every design decision we make, every criteria we apply, has to be validated against this single, uniform metric. This includes changes in the hardware architecture we prevalently use today, and it certainly involves new ways of writing software, starting from the operating system, runtime system, compilers, and at the application level. However the key point is that all those layers have to be co-designed, they are interdependent and cannot be seen as separate facets. The systems we have today have been evolving for over 50 years now. All layers function in a certain way relying on the other layers to do so as well. However, we do not have the time to wait for a coherent system to evolve for another 50 years. The new paradigms are needed now - therefore, co-design is the key.

## Governing Principles applied while Developing *HPX*

As it turn out, we do not have to start from scratch. Not everything has to be invented and designed anew. Many of the ideas needed to combat the 4 horsemen have already been had, often more than 30 years ago. All it takes is to gather them into a coherent approach. So please let me 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, for instance low latency network technologies like InfiniBand, caching memory hierarchies in all modern processors, the constant optimization of existing MPI implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use GPGPUs 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 to hide the latencies by filling the idle-time with useful work. Modern system 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 XMT). 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-preemptive, task-queue based parallelization solutions, such as Intel Threading Building Blocks (TBB), Microsoft Parallel Patterns Library (PPL), Cilk++, 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 Constrained Based Synchronization to replace Global Barriers

The code we write today is riddled with implicit (and explicit) global barriers. When I say global barrier I mean the synchronization of the control flow between several (very often all) threads (when using OpenMP) or processes (MPI). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP as the system synchronizes the threads used to execute the different iterations in parallel. In MPI each of the communication steps imposes an explicit barrier onto the execution flow as (often all) nodes have be synchronized. Each of those barriers acts as an eye of the needle the overall execution is forced to be squeezed through. Even minimal fluctuations in the execution times of the parallel threads (jobs) causes them to wait. Additionally it is often only one of the threads executing doing the actual reduce operation, which further impedes parallelism. A closer analysis of a couple of key algorithms used in science applications reveals that these global barriers are not always necessary. In many cases it is sufficient to synchronize a small subset of the threads. Any operation should proceed whenever the preconditions for its execution are met, and only those. Usually there is no need to wait for iterations of a loop to finish before you could continue calculating other things, all you need is to have those iterations done which were producing the required results for a particular next operation. Good bye global barriers, hello constraint based synchronization! People have been trying to build this type of computing (and even computers) already back in the 1970's. The theory behind what they did is based on ideas around static and dynamic dataflow. There are certain attempts today to get back to those ideas and to incorporate them with modern architectures. For instance, a lot of work is being done in the area of constructing dataflow oriented execution trees. Our results show that employing dataflow techniques in combination with the other ideas, as outlined herein, considerabley 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 PGAS (Partitioned Global Address Space) designed to overcome this limitation, such as Chapel, X10, UPC, or Fortress. However all systems based on PGAS rely on static data distribution. This works fine as long as such a static data distribution does not result in inhomogeneous 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++. The first attempts towards solving related problem go back decades as well, a good example is the Linda coordination language. 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 best performance it seems obvious to minimize the amount of bytes transferred from one part of the system to another. This is true on all levels. At the lowest level we try to take advantage of processor memory caches, thus minimizing memory latencies. Similarly, we try to amortize the data transfer time to and from GPGPUs as much as possible. At high levels we try to minimize data transfer between different nodes of a cluster or between different virtual machines on the cloud. Our experience (well, it's almost common wisdom) show that the amount of bytes necessary to encode a certain operation is very often much smaller than the amount of bytes encoding the data the operation is performed upon. Nevertheless we still often transfer the data to a particular place where we execute the operation just to bring the data back to where it came from afterwards. As an example let me look at the way we usually write our applications for clusters using MPI. This programming model is all about data transfer between nodes. MPI is the prevalent programming model for clusters, it is fairly straightforward to understand and to use. Therefore, we often write the applications in a way accommodating this model, centered around data transfer. These applications usually work well for smaller problem sizes and for regular data structures. The larger the amount of data we have to churn and the more irregular the problem domain becomes, the worse are the overall machine utilization and the (strong) scaling characteristics. While it is not impossible to implement more dynamic, data driven, and asynchronous applications using MPI, it is overly difficult to so. At the same time, if we look at applications preferring to execute the code close the locality where the data was placed, i.e. utilizing active messages (for instance based on Charm++), 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 to explicitly code the algorithms around the required communication scheme. As a result, any more than trivial MPI application spends a considerable amount of time waiting for incoming messages, thus causing starvation and latencies to impede full resource utilization. The more complex and more dynamic the data structures and algorithms become, the larger are the adverse effects. The community has discovered message-driven and (data-driven) methods of implementing algorithms a long time ago, and systems such as Charm++ already have integrated active messages demonstrating the validity of the concept. Message driven computation allows to send messages without that the receiver has 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 to almost completely overlap any communication with useful work, reducing latencies to a minimum.

## Examples

The following sections of our tutorial analyzes 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, yet powerful, *HPX* concepts.

**Note**

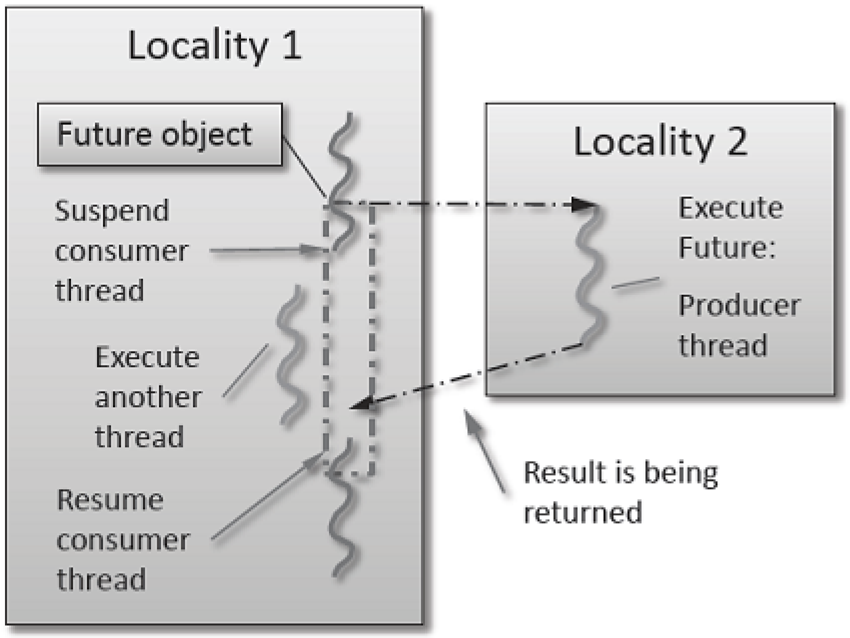
The instructions for building and running the examples currently only cover Unix variants.

## Fibonacci

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

As shown in the figure 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 action associated with the Future. The arguments of the action are bound when the Future is created.

**Figure 1. Schematic of a Future execution**



Once the action 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 action. When the result of the delayed computation is needed, a read operation is performed on the Future. If the Future's action 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.cpp.

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

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

To run the program type:

```
./bin/fibonacci
```

This should print (time should be approximate):

```
fibonacci(10) == 55
elapsed time: 0.00186288 [s]
```

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

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

Will yield:

```
fibonacci(20) == 6765
elapsed time: 0.233827 [s]
```

## Walkthrough

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 for command line processing. You can see that our programs `--n-value` option is set by calling the `add_options()` method on an instance of `boost::program_options::options_description`. The default value of the variable is set to 10. This is why when we ran the program for the first time without using the `--n-value` option the program returned the 10th value of the Fibonacci sequence. The constructor argument of the description is the text that appears when a user uses the `--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 or the Fibonacci Example for more details). `hpx::init()` calls `hpx_main()` after setting up *HPX*, which is where the logic of our program is encoded.

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

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

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

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

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

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

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

        char const* fmt = "fibonacci(%1%) == %2%\nelapsed time: %3% [s]\n";
        std::cout << (boost::format(fmt) % n % r % t.elapsed());
    }
    return hpx::finalize(); // Handles HPX shutdown
}
```

Upon a closer look we see that the hpx::lcos::future<> we created is assigned the return of hpx::lcos::async<fibonacci_action>(hpx::find_here(), n). hpx::lcos::async<>() takes an action, in this case fibonacci_action, and asynchronously kicks of the computation of the action, returning a future which represents the result of the computation. 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 hpx::lcos::async<>() 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
boost::uint64_t fibonacci(boost::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 the Accumulator Example). 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 aruments: 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 spawned by hpx::lcos::async<>(), which returns a future. Now, lets look at the function fibonacci():

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

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

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

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

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

This block of code is much more straightforward. 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, then we spawn two futures, n1 and n2. Each of these futures represents an asynchronous, recursive call to fibonacci(). After we've created both futures, we wait for both of them to finish computing, and then we 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.

## Hello World

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

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

### Setup

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

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

```
make examples.quickstart.hello_world
```

To run the program type:

```
./bin/hello_world
```

This should print:

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

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

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

will yield:

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

Notice how the ordering of the two print statements will change with subsequent runs. To run this program on multiple localities please see the section Using PBS.

### Walkthrough

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

Here is the main entry point. By using the include 'hpx/hpx_main.hpp' HPX will invoke the plain old C-main() as its first HPX thread.

```
int main()
{
    // Get a list of all available localities.
    std::vector<hpx::naming::id_type> localities =
        hpx::find_all_localities();

    // Reserve storage space for futures, one for each locality.
    std::vector<hpx::lcos::future<void> > futures;
    futures.reserve(localities.size());
    BOOST_FOREACH(hpx::naming::id_type const& node, localities)
    {
        // Asynchronously start a new task. The task is encapsulated in a
        // future, which we can query to determine if the task has
        // completed.
        typedef hello_world_foreman_action action_type;
        futures.push_back(hpx::async<action_type>(node));
    }

    // The non-callback version of hpx::lcos::wait takes a single parameter,
    // a future of vectors 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::lcos::wait()` 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::lcos::wait()`, 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 the Fibonacci Example, we used `hpx::find_here()` to specified 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 the Fibonacci Example our futures are set using `hpx::lcos::async<>()`. The `hello_world_foreman_action` is declared here:

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

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

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

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

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

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

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

        BOOST_FOREACH(std::size_t worker, attendance)
        {
            // Asynchronously start a new task. The task is encapsulated in a
            // future, which we can query to determine if the task has
            // completed.
            typedef hello_world_worker_action action_type;
            futures.push_back(hpx::async<action_type>(here, worker));
        }

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

Now, before we discuss `hello_world_foreman()`, let's talk about the `hpx::lcos::wait()` function. `hpx::lcos::wait()` provides a way to make sure that all of the futures have finished being calculated without having to call `hpx::lcos::future::get()` for each one. The version of `hpx::lcos::wait()` used here performs an non-blocking wait, which acts on an `std::vector<>`. It queries the state of the futures, waiting for them to finish. Whenever a future becomes marked as ready, `hpx::lcos::wait()` invokes a callback function provided by the user, supplying the callback function with the index of the future in the `std::vector<>` and the result of the future.

In `hello_world_foreman()`, an `std::set<>` called `attendance` keeps track of which OS-threads have printed out the hello world message. When the OS-thread prints out the statement, the future is marked as ready, and `hpx::lcos::wait()` invokes the callback function, in this case a C+11 lambda. This lambda erases the OS-threads id from the set `attendance`, thus letting `hello_world_foreman()` know which OS-threads still need to print out hello world. However, if the future returns a value of -1, the future executed on an OS-thread which has already printed out hello world. In this case, we have to try again by rescheduling the future in the next round. We do this by leaving the OS-thread id in `attendance`.

Finally, let us look at `hello_world_worker()`. Here, `hello_world_worker()` checks to see if it is on the target OS-thread. If it is executing on the correct OS-thread, it prints out the hello world message and returns the OS-thread id to `hpx::lcos::wait()` 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%";

        hpx::cout << (boost::format(msg) % desired % hpx::get_locality_id())
                  << std::endl << hpx::flush;

        return desired;
    }

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

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

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

## Accumulator

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

Components are globally named, meaning that a component action can be called remotely (e.g. from another machine). There are two accumulator examples in *HPX*; managed_accumulator and simple_accumulator (we will talk more about the differences between the two later). This tutorial will examine the managed_accumulator variant.

In the Fibonacci Example and the Hello World Example, 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: accumulators.

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

```
make examples.accumulator.managed_accumulator
```

To run the program type:

```
./bin/managed_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 managed_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: managed_accumulator_client.cpp.

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

- **A server class** - The implementation of the components functionality.
- **A stubs class** - A lower-level interface to instances of the component.
- **A client class** - A high-level interface that acts as a proxy for an instance of the component.

Typically, these three classes all have the same name, but stubs and server classes usually live in different sub-namespaces (`server` and `stubs` respectively). For example, the full names of the three classes in managed_accumulator are:

- `examples::server::managed_accumulator` (server class)
- `examples::stubs::managed_accumulator` (stubs class)
- `examples::managed_accumulator` (client class)

## The Server Class

The following code is from: server/managed_accumulator.hpp.

All *HPX* component server classes must inherit publicly from an *HPX* component base class. There are currently two component base classes:

- `hpx::components::managed_component_base<>` - Managed components are components which are allocated in bulk by *HPX*. Managed components are more efficient if you are creating a large number (e.g. hundreds or more per machine) of component instances.

- `hpx::components::simple_component_base<>` - Simple components are components which are allocated individually by *HPX*. Simple components are more efficient if you are creating a small number (e.g. only a handful per machine) of component instances.

The managed_accumulator component inherits from `hpx::components::locking_hock<>`. 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, managed_accumulator component is a managed component, because it also inherits from `hpx::components::managed_component_base<>` (the template argument passed to locking_hook is used as its base class). The following snippet shows the corresponding code:

```
class managed_accumulator
  : public hpx::components::locking_hook<
        hpx::components::managed_component_base<managed_accumulator>
    >
```

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

```
private:
    argument_type value_;
```

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

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

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

```
/// Reset the value to 0.
void reset()
{
    // set value_ to 0.
    value_= 0;
}

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

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

Here are the action types. These types wrap the methods we're exposing. The wrapping technique is very similar to the one used in the Fibonacci Example and the Hello World Example:

```
HPX_DEFINE_COMPONENT_ACTION(managed_accumulator, reset);
HPX_DEFINE_COMPONENT_ACTION(managed_accumulator, add);
HPX_DEFINE_COMPONENT_CONST_ACTION(managed_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::managed_accumulator::reset_action,
    managed_accumulator_reset_action);

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::managed_accumulator::add_action,
    managed_accumulator_add_action);

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::managed_accumulator::query_action,
    managed_accumulator_query_action);
```

**Note**

The code above must be placed in the global namespace.

The rest of the registration code is in managed_accumulator.cpp.

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

///////////////////////////////////////////////////////////////////////////////
typedef hpx::components::managed_component<
    examples::server::managed_accumulator
> accumulator_type;

HPX_REGISTER_MINIMAL_COMPONENT_FACTORY(accumulator_type, managed_accumulator);

///////////////////////////////////////////////////////////////////////////////
// Serialization support for managed_accumulator actions.
HPX_REGISTER_ACTION(
    accumulator_type::wrapped_type::reset_action,
    managed_accumulator_reset_action);
HPX_REGISTER_ACTION(
    accumulator_type::wrapped_type::add_action,
    managed_accumulator_add_action);
HPX_REGISTER_ACTION(
    accumulator_type::wrapped_type::query_action,
    managed_accumulator_query_action);
```

**Note**

The code above must be placed in the global namespace.

### The Stubs Class

The following code is from stubs/managed_accumulator.hpp.

All stubs classes must inherit from the stubs base class, `hpx::components::stub_base<>`:

```
struct managed_accumulator
  : hpx::components::stub_base<server::managed_accumulator>
```

The stubs class contains helper functions which invoke actions on component instances. There are a few different ways of invoking actions:

- **Non-blocking**: For actions which don't have return types, or when we do not care about the result of an action, we can invoke the action using fire-and-forget semantics. This means that once we have asked *HPX* to compute the action, we forget about it completely and continue with our computation. We use `hpx::applier::apply<>()` instead of `hpx::lcos::async<>()` to invoke an action in a non-blocking fashion. Here's an example from the managed_accumulator stubs class:

```
static void reset_non_blocking(hpx::naming::id_type const& gid)
{
    typedef server::managed_accumulator::reset_action action_type;
    hpx::apply<action_type>(gid);
}
```

- **Asynchronous**: Futures, as demonstrated in Fibonacci Example and the Hello World Example, enable asynchronous action invocation. Here's an example from the managed_accumulator stubs class:

```
static hpx::lcos::future<argument_type>
query_async(hpx::naming::id_type const& gid)
{
    typedef server::managed_accumulator::query_action action_type;
    return hpx::async<action_type>(gid);
}
```

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

```
static void
add_sync(hpx::naming::id_type const& gid, argument_type arg)
{
    typedef server::managed_accumulator::add_action action_type;
    hpx::async<action_type>(gid, arg).get();
}
```

`hpx::naming::id_type` is a type which represents a global identifier in *HPX*. This is the type that is returned by `hpx::find_here()`. This type specifies the target of an action.

## The Client Class

The following code is from managed_accumulator.hpp.

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

```
examples::managed_accumulator c;
c.create(hpx::find_here()); // Create a component on this machine.
```

and to invoke component actions:

```
c.add_sync(4);
```

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

```
class managed_accumulator
  : public hpx::components::client_base<
        managed_accumulator, stubs::managed_accumulator
    >
```

For readability, we typedef the base class like so:

```
typedef hpx::components::client_base<
    managed_accumulator, stubs::managed_accumulator
> base_type;
```

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

```
void reset_non_blocking()
{
    HPX_ASSERT(this->get_gid());
    this->base_type::reset_non_blocking(this->get_gid());
}
```

```
hpx::lcos::future<argument_type> query_async()
{
    HPX_ASSERT(this->get_gid());
    return this->base_type::query_async(this->get_gid());
}
```

```
void add_sync(argument_type arg)
{
    HPX_ASSERT(this->get_gid());
    this->base_type::add_sync(this->get_gid(), arg);
}
```

Note that `this->gid_` references a data member of the `hpx::components::client_base<>` base class.

## Interest Calculator

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

```
F = P(1 + i) ^ n
where:
    F= Future value
    P= Principal
    i= Interest rate
    n= number of compound periods
```

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

```
I = P * i
 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 Getting Started for information on configuring and building *HPX*) and enter:

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

To run the program type:

```
./bin/interest_calculator ---principal 100 ---rate 5 ---cp 6 ---time 36
```

This should print:

```
Final amount: 134.01
Amount made: 34.0096
```

## Walkthrough

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

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

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

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

Next we look at hpx_main.

```
int hpx_main(variables_map & vm)
{
    {
        using hpx::lcos::dataflow;
        using hpx::lcos::dataflow_base;
        hpx::naming::id_type here = hpx::find_here();
        double init_principal=vm["principal"].as<double>(); //Initial principal
        double init_rate=vm["rate"].as<double>(); //Interest rate
        int cp=vm["cp"].as<int>(); //Length of a compound period
        int t=vm["time"].as<int>(); //Length of time money is invested

        init_rate/=100; //Rate is a % and must be converted
        t/=cp; //Determine how many times to iterate interest calculation:
               //How many full compund 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!

        dataflow_base<double> principal = dataflow<identity_action>(here, init_principal);
        dataflow_base<double> rate = dataflow<identity_action>(here, init_rate);

        for (int i = 0; i < t; ++i)
        {
            dataflow_base<double> interest = dataflow<calc_action>(here, principal, rate);
            principal = dataflow<add_action>(here, principal, interest);
        }

        // wait for the dataflow execution graph to be finished calculating our
        // overall interest
        double result = principal.get_future().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 dataflows are set up. Notice that we first place our principal and rate into a dataflow by passing the variables `p` and `i_rate` to an action called `identity_action`:

```
// This is a helper function allowing to encapsulate the initial values into a
// dataflow object
double identity(double initial_value)
{
    return initial_value;
}

HPX_PLAIN_ACTION(identity, identity_action);
```

In this way `hpx::lcos::dataflow_base<double>` `principal` and `rate` will be initialized to `p` and `i_rate` when `hpx::lcos::dataflow<identity_action>` returns a value. These dataflows enter for loop and are passed to `interest`. Next `principal` and `interest` are passed to the reassignment of `principal`. 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;
}

///////////////////////////////////////////////////////////////////////////////
// Action Declarations
HPX_PLAIN_DIRECT_ACTION(calc, calc_action);
HPX_PLAIN_ACTION(add, add_action);
```

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

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

This statement calls `hpx::lcos::future::get()` on the last dataflow created 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.

# Manual

## Building *HPX* with CMake

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

### Introduction

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

If you are really anxious about getting a functional *HPX* build, go to the Quick Start section. If you are a CMake novice, start on Basic CMake Usage and then go back to the Quick Start once you know what you are doing. The Options and Variables section is a reference for customizing your build. If you already have experience with CMake, this is the recommended starting point.

### Quick Start

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

1. Download and install CMake here: CMake Downloads. Version 2.8 is the minimally required version for *HPX*.
2. Open a shell. Your development tools must be reachable from this shell through the PATH environment variable.
3. Create a directory for containing the build. It is not supported to build *HPX* on the source directory. cd to this directory:

```
mkdir mybuilddir
cd mybuilddir
```

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

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

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

This can fail if CMake can't detect your toolset, or if it thinks that the environment is not sane enough. On this case make sure that the toolset that you intend to use is the only one reachable from the shell and that the shell itself is the correct one for you development environment. CMake will refuse to build MinGW makefiles if you have a POSIX shell reachable through the PATH environment variable, for instance. You can force CMake to use a given build tool, see the Basic CMake Usage section.

**Caution**

The `ccmake` user interface program which wraps `cmake` and visualizes the current settings of the file `CMakeCache.txt` (that is where `cmake` stores all build settings) can be invoked with command line arguments in the same way as `cmake`. However, invoking `ccmake` this way does yield different results than the equivalent `cmake` invocation. This is a problem in `ccmake` which can cause subtle errors while building *HPX*.

Our suggestion is not use command line arguments to `ccmake` to alter the build configuration. Use the `cmake` command instead.

## Basic CMake Usage

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

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

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

```
cmake ---help
```

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

```
cmake --G -"Visual Studio 9 2008" path/to/hpx
```

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

## Options and Variables

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

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

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

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

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

Variables are listed here appending its type after a colon. It is correct to write the variable and the type on the CMake command line:

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

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

## Frequently used CMake Variables

Here are listed some of the CMake variables that are used often, along with a brief explanation and *HPX*-specific notes. For full documentation, check the CMake docs or execute `cmake --help-variable VARIABLE_NAME`.

### Frequently used CMake Variables

| | |
|---|---|
| `CMAKE_BUILD_TYPE:STRING` | Sets the build type for make based generators. Possible values are `Release`, `Debug`, `RelWithDebInfo` and `MinSizeRel`. The default value for this variable is `Release`. |
| `CMAKE_INSTALL_PREFIX:PATH` | Path where *HPX* will be installed if `make install` is invoked or the `INSTALL` target is built. The default value for this variable is `/opt/hpx` (Linux), or `C:/Program Files/hpx` (Windows). |

| | |
|---|---|
| `CMAKE_C_FLAGS:STRING` | Extra flags to use when compiling C source files. |
| `CMAKE_CXX_FLAGS:STRING` | Extra flags to use when compiling C++ source files. |
| `CMAKE_FORTRAN_FLAGS:STRING` | Extra flags to use when compiling Fortran source files. |
| `CMAKE_VERBOSE_MAKEFILE:BOOL` | Create verbose makefiles if `ON` (default: `OFF`). CMake will produce verbose makefiles that show each command line as it is used. |

## *HPX* specific CMake Variables

### Frequently used *HPX* specific CMake Variables

Here are listed the most frequently used CMake variables specific to *HPX*, along with a brief explanation.

#### Frequently used __hpx__ specific CMake Variables

| | |
|---|---|
| `HPX_BUILD_DOCUMENTATION:BOOL` | Build the *HPX* documentation if the documentation toolchain is available (default: `OFF`). For more information about how to set up the documentation tool chain please refer to the section Setting up the HPX Documentation Tool Chain. |
| `HPX_BUILD_EXAMPLES:BOOL` | *HPX* examples will be built (default: `ON`). To actually build the examples execute `make examples`. |
| `HPX_BUILD_TESTS:BOOL` | *HPX* tests will be built (default: `ON`). To actually build the tests execute `make tests`. |
| `HPX_CMAKE_LOGLEVEL:STRING` | Sets the debugging level for the *HPX* build system generation. Possible values are `Error`, `Warn`, `Debug`, and `Info`. The default is `Warn`. |
| `HPX_NO_LOGGING:BOOL` | Sets whether *HPX* should not support generating detailed logging output (default: `OFF`). Even if this is set to `OFF` (i.e. logging is supported) you need to enable logging when running any *HPX* application. Logging support introduces a minimal runtime overhead which usually can be ignored. This option removes runtime overhead but also disables the capability to extract detailed runtime error information. |
| `HPX_HAVE_CXX11:BOOL` | Sets whether *HPX* should use C++11 support, if available (default: `ON`). |
| `HPX_HAVE_STACKTRACES:BOOL` | Sets whether *HPX* applications should support capturing stack back- traces (default: `ON`). |
| `HPX_HAVE_NATIVE_TLS:BOOL` | Sets whether *HPX* should use the native compiler support for thread local storage, if available (default: `ON`). |
| `HPX_THREAD_DEBUG_INFO:BOOL` | Sets whether *HPX* threads should maintain detailed debugging information like the parent thread id, the phase execution count, the thread description, and the LCO description (while the thread is suspended). The default value for this variable is `OFF` if `CMAKE_BUILD_TYPE=Release` and `ON` otherwise. |
| `HPX_NO_INSTALL:BOOL` | Sets whether *HPX* can be used directly from the build directory. (default: OFF) |
| `HPX_NATIVE_MIC:BOOL` | Build *HPX* to run natively on the Xeon Phi Coprocessor (default: OFF) |

### Setting the memory Allocator

The following list is about the default memory allocator used by *HPX*. We currently support the regular system allocator, tcmalloc, jemalloc and tbbmalloc. The allocator is crucial in getting an application to scale with *HPX*. The default system allocator is usually a point of heavy contention due to syscalls and locks in the operating system. This can be avoided by using the above custom allocators. For using *HPX* on the Xeon Phi we only support system and tbbmalloc. tbbmalloc is part of the Intel Thread Building Blocks.

### Affecting the default memory allocator

| | |
|---|---|
| `HPX_MALLOC:STRING` | Set the memory allocator to use. Defaults to tcmalloc on regular systems and to tbbmalloc if HPX_NATIVE_MIC is set to On. If the specific memory allocator isn't found we silently fall back to the system allocator. |

### Locations for memory allocators

| | |
|---|---|
| `TCMALLOC_ROOT:PATH` | Specifies where to look at for tcmalloc |
| `JEMALLOC_ROOT:PATH` | Specifies where to look at for jemalloc |
| `TBB_ROOT:PATH` | Specifies where to look at for tbb. tbbmalloc is part of tbb |

## Variables concerning Thread Scheduling Policies

Here is a list of options controlling the scheduling policies available for all applications built with *HPX*. For more information about the avalable scheduling policies see the section *HPX* Thread Scheduling Policies.

### Enable __hpx__ Scheduling Policies

| | |
|---|---|
| `HPX_STATIC_PRIORITY_SCHEDULER:BOOL` | Enable static priority scheduling policy (default: `OFF`) |
| `HPX_STATI_SCHEDULER:BOOL` | Enable local scheduling policy (default: `OFF`) |
| `HPX_GLOBAL_SCHEDULER:BOOL` | Enable global scheduling policy (default: `OFF`) |
| `HPX_ABP_SCHEDULER:BOOL` | Enable ABP scheduling policy (default: `OFF`) |
| `HPX_ABP_PRIORITY_SCHEDULER:BOOL` | Enable APB priority scheduling policy (default: `OFF`) |
| `HPX_HIERARCHY_SCHEDULER:BOOL` | Enable hierarchy scheduling policy (default: `OFF`) |
| `HPX_PERIODIC_PRIORITY_SCHEDULER:BOOL` | Enable periodic priority scheduling policy (default: `OFF`) |

## Variables concerning Parcelports

Here is a list of options controlling the parcelports available for all applications built with *HPX*.

### Enable __hpx__ Parcelports

| | |
|---|---|
| `HPX_HAVE_PARCELPORT_IPC:BOOL` | Enable parcelport based on shared memory. This is only available if you use a boost version greater than 1.51 (default: OFF) |
| `HPX_HAVE_PARCELPORT_IBVERBS:BOOL` | Enable parcelport based on rdma ibverbs operations. We use rdmacm to establish our connections. In order use the parcelport, please set `IBVERBS_ROOT` and `RDMACM_ROOT` to point to the appropriate locations of your ofed stack. (default: OFF) |
| `HPX_HAVE_PARCELPORT_MPI:BOOL` | Enable parcelport based on MPI. Please set `MPI_CXX_COMPILER` and `MPI_C_COMPILER` to point to the correct location of your MPI compiler wrappers (default: OFF) |
| `HPX_PARCELPORT_MPI_ENV:LIST` | Additional environment variable names for automatic MPI detection. By default HPX checks for PMI_RANK (Intel MPI and MVAPICH2) and OMPI_COMM_WORLD_SIZE (OpenMPI) |

### Search Paths for the ibverbs parcelport

| | |
|---|---|
| `IBVERBS_ROOT` | Path to the root directory where the ibverbs libraries and headers can be found |

|  |  |
|---|---|
| `RDMACM_ROOT` | Path to the root directory where the rdmacm library and headers can be found |

## Search Paths for the MPI parcelport

|  |  |
|---|---|
| `MPI_CXX_COMPILER` | Path to the mpi C++ wrapper compiler script |

Here is a more complete list of CMake variables specific to *HPX*. These variables are used more seldom and are mainly useful for debugging purposes.

## Other __hpx__ specific CMake Variables

|  |  |
|---|---|
| `HPX_THREAD_MAINTAIN_BACKTRACE_ON_SUSPENSION:BOOL` | Sets whether *HPX* threads should capture the stack back-trace (list of stack frames) whenever they are about to be suspended (default: `OFF`). |
| `HPX_THREAD_MAINTAIN_FULLBACKTRACE_ON_SUSPENSION:BOOL` | Sets whether *HPX* threads should capture the stack back-trace as a string whenever they are about to be suspended (default: `OFF`). If this option is enabled *HPX* will not store the stack back-trace (list of stack frames) as enabled by `HPX_THREAD_MAINTAIN_BACKTRACE_ON_SUSPENSION`. |
| `HPX_HAVE_VERIFY_LOCKS:BOOL` | Sets whether *HPX* should be configured to enable code which verifies that no locks are being held during thread suspension (default: `OFF`). This option is mostly useful for debugging purposes as locks which are being held during thread suspension can easily cause deadlocks. |
| `HPX_HAVE_VERIFY_LOCKS_BACKTRACE:BOOL` | Sets whether *HPX* should be configured to store the stack back-trace as a string for are locks which are registered with the lock tracking (default `OFF`). This allows to track at what place a lock was acquired. |
| `HPX_THREAD_BACKTRACE_ON_SUSPENSION_DEPTH:STRING` | Sets the depth of the stack back-traces captured during thread suspension (default: `5`). This value is only meaningful if the CMake variable `HPX_THREAD_MAINTAIN_BACKTRACE_ON_SUSPENSION` is set to `ON`. |
| `HPX_USE_ITT_NOTIFY:BOOL` | Sets whether *HPX* supports integration with the diagnostic tools of the Intel Parallel Studio (Intel Amplifier and Intel Inspector). The default value for this variable is `OFF`. Even if the variable is set to `ON` you must separately enable integration with the Intel tools at runtime. This option is available only if the include files and libraries for one of the Intel tools can be located (see CMake variable `AMPLIFIER` below). |
| `HPX_AUTOMATIC_PREPROCESSING:BOOL` | Sets whether the generated makefiles should regenerate the partially preprocessed files that are part of *HPX*. The default value for this option is `OFF`. This option can be enabled only if the Boost Wave tool is available (see CMake variable `BOOSTWAVE` below). |
| `HPX_HAVE_COMPRESSION_ZLIB:BOOL` | Sets whether support for compressing parcels using the ZLib library will be enabled or not. This variable is set to `OFF` by default. If you enable this option CMake will try to find the ZLib library, which might require setting the CMake variable `ZLIB_ROOT`. |
| `HPX_HAVE_COMPRESSION_BZIP2:BOOL` | Sets whether support for compressing parcels using the BZip2 library will be enabled or not. This variable is set to `OFF` by default. If you enable this option CMake will try to find the BZip2 library, which might require setting the CMake variable `BZIP_ROOT`. |
| `HPX_HAVE_COMPRESSION_SNAPPY:BOOL` | Sets whether support for compressing parcels using the Snappy library will be enabled or not. This variable is set to `OFF` by default. If you enable this option CMake will try to find the Snappy library, which might require setting the CMake variable `SNAPPY_ROOT`. |
| `HPX_USE_MORE_THAN_64_THREADS:BOOL` | Sets whether *HPX* should be configured to run more than 64 threads (for system having more than 64 processing units, like the XeonPhi device). The variable is set to `OFF` by default. |
| `HPX_HAVE_SECURITY:BOOL` | Sets whether *HPX* should be configured to include security related code which currently secures inter-locality related traffic and inter-locality action invocations. |

## Additional Tools and Libraries used by HPX

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

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

### Additional Tools and Libraries used by HPX

| | |
|---|---|
| `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 a 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 Portable Hardware Locality (HWLOC) library. While it is not necessary to compile *HPX* with HWLOC, we strongly suggest you do so. 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 Networking and Cryptography library (NaCl) library. The PAPI library is necessary to compile a special component exposing PAPI hardware events and counters as *HPX* performance counters. This is not available on the Windows platform. |
| `BOOSTWAVE_ROOT:PATH` | Specifies where to look for a built Boost Wave binary. Set this if you want to recreate the preprocessed header files used by *HPX*. Normally this will not be necessary, unless you modified the sources those preprocessed headers depend on. |
| `AMPLIFIER_ROOT:PATH` | Specifies where to look for one of the tools of the Intel Parallel Studio(tm) product, either Intel Amplifier(tm) or Intel Inspector(tm). This should be set if the CMake variable `HPX_USE_ITT_NOTIFY` is set to `ON`. Enabling ITT support in *HPX* will integrate any application with the mentioned Intel tools, which customizes the generated information for your application and improves the generated diagnostics. |
| `SODIUM_ROOT:PATH` | Specifies where to look for the Performance Application Programming Interface (PAPI) library. The Sodium library is necessary to enable the security related functionality (see `HPX_HAVE_SECURITY`). |

### Additional Tools and Libraries Required by some of the Examples

| | |
|---|---|
| `HDF5_ROOT:PATH` | Specifies where to look for the Hierarchical Data Format V5 (HDF5) include files and libraries. |
| `MPI_ROOT:PATH` | Specifies where to look for the MPI include files and libraries. |
| `MPIEXEC_ROOT:PATH` | Specifies where to look for the MPI tool `mpiexec`. |

## Setting up the *HPX* Documentation Tool Chain

The documentation for *HPX* is generated by the Boost QuickBook documentation toolchain. Setting up this toolchain requires to install several tools and libraries. Generating the documentation is possible only if all of those are configured correctly.

### CMake Variables needed for the Documentation Toolchain

| | |
|---|---|
| `DOXYGEN_ROOT:PATH` | Specifies where to look for the installation fo the Doxygen tool. |
| `BOOSTQUICKBOOK_ROOT:PATH` | Specifies where to look for the installation fo the QuickBook tool. This tool usually needs to be built by hand. See the QuickBook documentation for more details on how to do this. |

| | |
|---|---|
| `BOOSTAUTOINDEX_ROOT:PATH` | Specifies where to look for the installation fo the AutoIndex tool. This tool usually needs to be built by hand. See the AutoIndex documentation for more details on how to do this. The documentation can still be generated even if the AutoIndex tool cannot be found. |
| `XSLTPROC_ROOT:PATH` | Specifies where to look for the installation of the libxslt package (and the xsltproc tool). Consult the documentation for your platform on how to make this package available on your machine. |
| `DOCBOOK_DTD_ROOT:PATH` | Specifies where to look for the installation of the docbook-xml-4.2 package. This usually needs to refer to the directory containing the file `docbook.cat`, which is part of this package. |
| `DOCBOOK_XSL_ROOT:PATH` | Specifies where to look for the installation of the docbook-xsl package. This usually needs to refer to the directory containing the file `catalog.xml`, which is part of this package. |

## 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` *`HPX`* `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/include/iostreams.hpp>

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

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

Now, in the directory where you put hello_world.cpp, issue the following commands (where `$HPX_LOCATION` is the `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` --
liostreams --DHPX_APPLICATION_NAME=hello_world
```

**Important**

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

**Note**

If *HPX* was build in debug mode (`cmake -DCMAKE_BUILD_TYPE=Debug`) the pkg-config names have to be different. 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 `-liostreams` you will need to specify `-liostreamsd`.

To test the program, type:

```
./hello_world
```

which should print `Hello World!` and exit.

## How to Build *HPX* Components with pkg-config

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

**hello_world_component.cpp**

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

namespace examples { namespace server
{

void hello_world::invoke()
{
    hpx::cout << "Hello HPX World!\n" << hpx::flush;
}

}}

HPX_REGISTER_COMPONENT_MODULE();

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

HPX_REGISTER_MINIMAL_COMPONENT_FACTORY(hello_world_type, hello_world);

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

**hello_world_component.hpp**

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

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

namespace examples { namespace server
{

struct HPX_COMPONENT_EXPORT hello_world
  : hpx::components::managed_component_base<hello_world>
{
    void invoke();
    HPX_DEFINE_COMPONENT_ACTION(hello_world, invoke);
};

}
namespace stubs
{

struct hello_world : hpx::components::stub_base<server::hello_world>
{
    static void invoke(hpx::naming::id_type const& gid)
    {
        hpx::async<server::hello_world::invoke_action>(gid).get();
    }
};

}

struct hello_world
  : hpx::components::client_base<hello_world, stubs::hello_world>
{
    typedef hpx::components::client_base<hello_world, stubs::hello_world>
        base_type;

    void invoke()
    {
        this->base_type::invoke(this->get_gid());
    }
};

}

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

#endif // HELLO_WORLD_COMPONENT_HPP
```

**hello_world_client.cpp**

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

int hpx_main(boost::program_options::variables_map&)
{
    {
        // Create a single instance of the component on this locality.
        examples::hello_world client =
            examples::hello_world::create(hpx::find_here());

        // Invoke the components action, which will print -"Hello World!".
        client.invoke();
    }

    return hpx::finalize(); // Initiate shutdown of the runtime system.
}

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

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

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

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

```
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig
c++ --o libhello_world.so hello_world_component.cpp `pkg-config ---cflags ---libs -
hpx_component` --DHPX_COMPONENT_NAME=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 libhello_world.so ~/my_hpx_libs
```

**Note**

If *HPX* was build in debug mode (`cmake -DCMAKE_BUILD_TYPE=Debug`) the pkg-config names have to be different. 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 `-liostreams` you will need to specify `-liostreamsd`.

In the ~/my_hpx_libs directory you need to create an ini file inside that directory which matches the name of the component (as supplied by -DHPX_COMPONENT_NAME above).

**hello_world.ini**

```
[hpx.components.hello_world]
name = hello_world
path = ${HOME}/my_hpx_libs
```

**Note**

For additional details about ini file configuration and *HPX*, see Loading INI Files

In addition, you'll need this in your home directory:

**.hpx.ini**

```
[hpx]
ini_path = $[hpx.ini_path]:${HOME}/my_hpx_libs
```

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

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

**Important**

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

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

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

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

## How to Use *HPX* with CMake

In Addition to the pkg-config support discussed on the previous pages, *HPX* comes with full CMake support. In order to integrate *HPX* into your existing, or new CMakeLists.txt you can include `FindHPX.cmake`. Following is the Hello World component example using CMake instead of pkg-config.

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

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

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

```
# Require a recent version of cmake
cmake_minimum_required(VERSION 2.8.4 FATAL_ERROR)

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

# This adds the HPX cmake configuration directory to the search path.
set(CMAKE_MODULE_PATH
    ${HPX_ROOT}/share/cmake-${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION}/Modules)

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

include_directories(${HPX_INCLUDE_DIR})
link_directories(${HPX_LIBRARY_DIR})
```

This cmake script assumes that the location of the *HPX* installation is set as the value of HPX_ROOT which is best done by passing it on the command line while invoking cmake:

```
cmake --DHPX_ROOT=<HPX installations directory> -...
```

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

The last two lines in the snippet above sets the appropriate include and link directories.

In order to add the *HPX* component we use the `add_hpx_component` macro and add it to the `CMakeLists.txt` file:

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

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
- `COMPILE_FLAGS`: Additional compiler flags
- `LINK_FLAGS`: Additional linker flags
- `ESSENTIAL`: Build this component as part of the `all` target

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

```
# build your application using HPX
add_hpx_executable(hello_world
    ESSENTIAL
    SOURCES hello_world_client.cpp
    COMPONENT_DEPENDENCIES hello_world_component)
```

When you configure your application, all you need to do is set the HPX_ROOT variable to point to the installation of *HPX*!

**Note**

`FindHPX.cmake` makes adjustments various cmake internal settings such as compiler and linker flags. This is needed to ensure maximal compatibility between your application and the *HPX* version you are using. The `add_hpx_executable` and `add_hpx_component` additionally add the proper libraries to link against.

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.

## Testing *HPX*

To ensure correcteness of *HPX*, we ship a large variety of unit and regression tests. The tests are driven by the CTest tool and are executed automatically by buildbot (see HPX Buildbot Website) on each commit to the *HPX* Github 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 to submit an issue on our *HPX* Issues tracker with detailed information about the target system.

### Running tests manually

Running the tests manually is as easy as typing

```
make tests
```

. This will build all tests and run them once the tests are build successfully. After the tests have been built, you can invoke seperate tests with the help of the

```
ctest
```

command. Please see the CTest Documentation for further details.

## Issue Tracker

If you stumble over a bug or missing feature missing feature in *HPX* please submit an issue to our *HPX* Issues. For more information on how to submit support requests or other means of getting in contact with the developers please see the Support Website.

## Buildbot

In addition to manual testing, we run automated tests on various platforms. You can see the status of the current master head by visiting the HPX Buildbot Website.

# Launching *HPX*

## Configure *HPX* Applications

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

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

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

### The *HPX* INI File Format

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

```
name=value
```

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

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

```
[section]
```

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

```
[outer_section.inner_section]
```

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

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

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

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

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

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

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

**Note**

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

## Built-in Default Configuration Settings

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

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

**Note**

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

### *The `system` Configuration Section*

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

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

***The hpx Configuration Section***

```
[hpx]
location = ${HPX_LOCATION:$[system.prefix]}
component_path = $[hpx.location]/lib/hpx:$[system.executable_prefix]/lib/hpx:
$[system.executable_prefix]/../lib/hpx
master_ini_path = $[hpx.location]/share/hpx-<version>:$[system.executable_prefix]/share/hpx-
<version>:$[system.executable_prefix]/../share/hpx-<version>
ini_path = $[hpx.master_ini_path]/ini
os_threads = 1
localities = 1
program_name =
cmd_line =
lock_detection = 0
minimal_deadlock_detection = <debug>

[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_USE_GUARD_PAGES:1}
```

| Property | Description |
|---|---|
| `hpx.location` | This is initialized to the id of the locality this application instance is running on. |
| `hpx.component_path` | This is initialized to the list of directories where the *HPX* runtime library will look for installed components. Duplicates are discarded. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows). |
| `hpx.master_ini_path` | This is initialized to the list of default paths of the main hpx.ini configuration files. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows). |
| `hpx.ini_path` | This is initialized to the default path where *HPX* will look for more ini configuration files. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows). |

| **Property** | **Description** |
|---|---|
| `hpx.os_threads` | This setting reflects the number of OS-threads used for running *HPX*-threads. Defaults to `1`. |
| `hpx.localities` | This setting reflects the number of localities the application is running on. Defaults to `1`. |
| `hpx.program_name` | This setting reflects the program name of the application instance. Initialized from the command line (`argv\[0\]`). |
| `hpx.cmd_line` | This setting reflects the actual command line used to launch this application instance. |
| `hpx.lock_detection` | This setting verifies that no locks are being held while a *HPX* thread is suspended. This setting is applicable only if `HPX_VERIFY_LOCKS` is set during configuration in CMake. |
| `hpx.minimal_deadlock_detection` | This setting enables support for minimal deadlock detection for *HPX*-threads. By default this is set to `1` (for Debug builds) or to `0` (for Release, RelWithDebInfo, RelMinSize builds), this setting is effective only if `HPX_THREAD_MINIMAL_DEADLOCK_DETECTION` is set during configuration in CMake. |
| `hpx.stacks.small_size` | This is initialized to the small stack size to be used by *HPX*-threads. Set by default to the value of the compile time preprocessor constant `HPX_SMALL_STACK_SIZE` (defaults to `0x8000`). |
| `hpx.stacks.medium_size` | This is initialized to the medium stack size to be used by *HPX*-threads. Set by default to the value of the compile time preprocessor constant `HPX_MEDIUM_STACK_SIZE` (defaults to `0x20000`). |
| `hpx.stacks.large_size` | This is initialized to the large stack size to be used by *HPX*-threads. Set by default to the value of the compile time preprocessor constant `HPX_LARGE_STACK_SIZE` (defaults to `0x200000`). |
| `hpx.stacks.huge_size` | This is initialized to the huge stack size to be used by *HPX*-threads. Set by default to the value of the compile time preprocessor constant `HPX_HUGE_STACK_SIZE` (defaults to `0x2000000`). |
| `hpx.stacks.use_guard_pages` | This entry controls whether the coroutine library will generate stack guard pages or not. This entry is applicable on Linux only and only if the `HPX_USE_GENERIC_COROUTINE_CONTEXT` option is not enabled and the `HPX_THREAD_GUARD_PAGE` is set to 1 while configuring the build system. It is set by default to `1`. |

#### *The `hpx.threadpools` Configuration Section*

```
[hpx.threadpools]
io_pool_size = ${HPX_NUM_IO_POOL_THREADS:2}
parcel_pool_size = ${HPX_NUM_PARCEL_POOL_THREADS:2}
timer_pool_size = ${HPX_NUM_TIMER_POOL_THREADS: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.components` Configuration Section*

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

| **Property** | **Description** |
|---|---|
| `hpx.components.load_external` | This entry defines whether external components will be loaded on this locality. This entry normally is set to `1` and usually there is no need to directly change this value. It is automatically set to `0` for a dedicated AGAS server locality. |

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

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

| **Property** | **Description** |
|---|---|
| `hpx.components.<component_instance_name>.name` | This is the name of a component, usually the same as the second argument to the macro used while registering the component with `HPX_REGISTER_MINIMAL_COMPONENT_FACTORY`. 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. |

| Property | Description |
|---|---|
| `hpx.components.<component_instance_name>.enabled` | This setting explicitly enables or disables the component. This is an optional property, *HPX* assumed that the component is enabled if it is not defined. |

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

### *The `hpx.parcel` Configuration Section*

```
[hpx.parcel]
address = ${HPX_PARCEL_SERVER_ADDRESS:<hpx_initial_ip_address>}
port = ${HPX_PARCEL_SERVER_PORT:<hpx_initial_ip_port>}
bootstrap = ${HPX_PARCEL_BOOTSTRAP:<hpx_parcel_bootstrap>}
max_connections = ${HPX_PARCEL_MAX_CONNECTIONS:<hpx_parcel_max_connections>}
max_connections_per_locality = -
${HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY:<hpx_parcel_max_connections_per_locality>}
max_message_size = ${HPX_PARCEL_MAX_MESSAGE_SIZE:<hpx_parcel_max_message_size>}
array_optimization = ${HPX_PARCEL_ARRAY_OPTIMIZATION:1}
zero_copy_optimization = ${HPX_PARCEL_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.array_optimization]}
async_serialization = ${HPX_PARCEL_ASYNC_SERIALIZATION:1}
enable_security = ${HPX_PARCEL_ENABLE_SECURITY:0}
```

| Property | Description |
|---|---|
| `hpx.parcel.address` | This property defines the default IP address to be used for the parcel layer to listen to. This IP address will be used as long as no other values are specified (for instance using the `--hpx:hpx` command line option). The expected format is any valid IP address or domain name format which can be resolved into an IP address. The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_ADDRESS` (`"127.0.0.1"`). |
| `hpx.parcel.port` | This property defines the default IP port to be used for the parcel layer to listen to. This IP port will be used as long as no other values are specified (for instance using the `--hpx:hpx` command line option). The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_PORT` (`7010`). |
| `hpx.parcel.bootstrap` | This property defines which parcelport type should be used during application bootstrap. The default depends on the compile time preprocessor constant `HPX_PARCEL_BOOTSTRAP` (`"tcp"`). |
| `hpx.parcel.max_connections` | This property defines how many network connections between different localities are overall kept alive by each of locality. The default depends on the compile time preprocessor constant `HPX_PARCEL_MAX_CONNECTIONS` (`512`). |
| `hpx.parcel.max_connections_per_locality` | This property defines the maximum number of network connections that one locality will open to another locality. The default depends on the compile time preprocessor constant `HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY` (`4`). |

| **Property** | **Description** |
|---|---|
| `hpx.parcel.max_message_size` | This property defines the maximum allowed message size which will be transferrable through the parcel layer. The default depends on the compile time preprocessor constant `HPX_PARCEL_MAX_MESSAGE_SIZE` (`1000000000`) bytes. |
| `hpx.parcel.array_optimization` | This property defines whether this locality is allowed to utilize array optimizations during serialization of parcel data. The default is `1`. |
| `hpx.parcel.zero_copy_optimization` | This property defines whether this locality is allowed to utilize zero copy optimizations during serialization of parcel data. The default is the same value as set for `hpx.parcel.array_optimization`. |
| `hpx.parcel.async_serialization` | This property defines whether this locality is allowed to spawn a new thread for serialization (this is both for encoding and decoding parcels). The default is `1`. |
| `hpx.parcel.enable_security` | This property defines whether this locality is encrypting parcels. The default is `0`. |

```
[hpx.parcel.tcp]
enable = ${HPX_HAVE_PARCELPORT_TCP:1}
array_optimization = ${HPX_PARCEL_TCP_ARRAY_OPTIMIZATION:$[hpx.parcel.array_optimization]}
zero_copy_optimization = ${HPX_PARCEL_TCP_ZERO_COPY_OPTIMIZATION:
$[hpx.parcel.zero_copy_optimization]}
async_serialization = ${HPX_PARCEL_TCP_ASYNC_SERIALIZATION:$[hpx.parcel.async_serialization]}
enable_security = ${HPX_PARCEL_TCP_ENABLE_SECURITY:$[hpx.parcel.enable_security]}
```

| **Property** | **Description** |
|---|---|
| `hpx.parcel.tcp.enable` | Enable the use of the default TCP parcelport. Note that the initial bootstrap of the overall *HPX* application will be performed using the default TCP connections. This parcelport is enabled by default. This will be disabled only if MPI is enabled (see below). |
| `hpx.parcel.tcp.array_optimization` | This property defines whether this locality is allowed to utilize array optimizations in the TCP/IP parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.array_optimization`. |
| `hpx.parcel.tcp.zero_copy_optimization` | This property defines whether this locality is allowed to utilize zero copy optimizations in the TCP/IP parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`. |
| `hpx.parcel.tcp.async_serialization` | This property defines whether this locality is allowed to spawn a new thread for serialization in the TCP/IP parcelport (this is both for encoding and decoding parcels). The default is the same value as set for `hpx.parcel.async_serialization`. |

| Property | Description |
|---|---|
| hpx.parcel.tcp.enable_security | This property defines whether this locality is encrypting parcels in the TCP/IP parcelport. The default is the same value as set for hpx.parcel.enable_security. |

The following settings relate to the shared memory parcelport (which is usable for communication between two localities on the same node). These settings take effect only if the compile time constant HPX_HAVE_PARCELPORT_IPC is set (the equivalent cmake variable is HPX_HAVE_PARCELPORT_IPC, and has to be set to ON).

```
[hpx.parcel.ipc]
enable = ${HPX_HAVE_PARCELPORT_IPC:0}
data_buffer_cache_size=${HPX_PARCEL_IPC_DATA_BUFFER_CACHE_SIZE:512}
array_optimization = ${HPX_PARCEL_IPC_ARRAY_OPTIMIZATION:$[hpx.parcel.array_optimization]}
async_serialization = ${HPX_PARCEL_IPC_ASYNC_SERIALIZATION:$[hpx.parcel.async_serialization]}
enable_security = ${HPX_PARCEL_IPC_ENABLE_SECURITY:$[hpx.parcel.enable_security]}
```

| Property | Description |
|---|---|
| hpx.parcel.ipc.enable | Enable the use of the shared memory parcelport for connections between localities running on the same node. Note that the initial bootstrap of the overall *HPX* application will still be performed using the default TCP connections. This parcelport is disabled by default. |
| hpx.parcel.ipc.data_buffer_cache_size | This property specifies the number of cached data buffers used for interprocess communication between localities on the same node. The default depends on the compile time preprocessor constant HPX_PARCEL_IPC_DATA_BUFFER_CACHE_SIZE (512). |
| hpx.parcel.ipc.array_optimization | This property defines whether this locality is allowed to utilize array optimizations in the shared memory parcelport during serialization of parcel data. The default is the same value as set for hpx.parcel.array_optimization. |
| hpx.parcel.ipc.async_serialization | This property defines whether this locality is allowed to spawn a new thread for serialization in the shared memory parcelport (this is both for encoding and decoding parcels). The default is the same value as set for hpx.parcel.async_serialization. |
| hpx.parcel.ipc.enable_security | This property defines whether this locality is encrypting parcels in the shared memory parcelport. The default is the same value as set for hpx.parcel.enable_security. |

The following settings relate to the Infiniband parcelport. These settings take effect only if the compile time constant HPX_HAVE_PARCELPORT_IBVERBS is set (the equivalent cmake variable is HPX_HAVE_PARCELPORT_IBVERBS, and has to be set to ON).

```
[hpx.parcel.ibverbs]
enable = ${HPX_HAVE_PARCELPORT_IBVERBS:0}
buffer_size = ${HPX_PARCEL_IBVERBS_BUFFER_SIZE:65536}
array_optimization = ${HPX_PARCEL_IBVERBS_ARRAY_OPTIMIZATION:$[hpx.parcel.array_optimization]}
async_serialization = ${HPX_PARCEL_IBVERBS_ASYNC_SERIALIZATION:$[hpx.parcel.async_serialization]}
enable_security = ${HPX_PARCEL_IBVERBS_ENABLE_SECURITY:$[hpx.parcel.enable_security]}
```

| **Property** | **Description** |
|---|---|
| `hpx.parcel.ibverbs.enable` | Enable the use of the ibverbs parcelport for connections between localities running on a node with infiniband capable hardware. Note that the initial bootstrap of the overall *HPX* application will still be performed using the default TCP connections. This parcelport is disabled by default. |
| `hpx.parcel.ibverbs.buffer_size` | This property specifies the size in bytes of the buffers registered to the infiniband hardware. Parcels which are smaller than this will be serialized and sent over the network in a zero-copy fashion. Parcels bigger than this will be transparently copied to a big enough temporary buffer. |
| `hpx.parcel.ibverbs.array_optimization` | This property defines whether this locality is allowed to utilize array optimizations in the ibverbs parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.array_optimization`. |
| `hpx.parcel.ibverbs.async_serialization` | This property defines whether this locality is allowed to spawn a new thread for serialization in the ibverbs parcelport (this is both for encoding and decoding parcels). The default is the same value as set for `hpx.parcel.async_serialization`. |
| `hpx.parcel.ibverbs.enable_security` | This property defines whether this locality is encrypting parcels in the ibverbs parcelport. The default is the same value as set for `hpx.parcel.enable_security`. |

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_HAVE_PARCELPORT_MPI`, and has to be set to `ON`).

```
[hpx.parcel.mpi]
enable = ${HPX_HAVE_PARCELPORT_MPI:1}
env = ${HPX_PARCELPORT_MPI_ENV:PMI_RANK,OMPI_COMM_WORLD_SIZE}
multithreaded = ${HPX_PARCELPORT_MPI_MULTITHREADED:0}
rank = <MPI_rank>
processor_name = <MPI_processor_name>
array_optimization = ${HPX_PARCEL_MPI_ARRAY_OPTIMIZATION:$[hpx.parcel.array_optimization]}
zero_copy_optimization = ${HPX_PARCEL_MPI_ZERO_COPY_OPTIMIZATION:
$[hpx.parcel.zero_copy_optimization]}
use_io_pool = ${HPX_PARCEL_MPI_USE_IO_POOL:$1}
async_serialization = ${HPX_PARCEL_MPI_ASYNC_SERIALIZATION:$[hpx.parcel.async_serialization]}
enable_security = ${HPX_PARCEL_MPI_ENABLE_SECURITY:$[hpx.parcel.enable_security]}
```

| **Property** | **Description** |
|---|---|
| `hpx.parcel.mpi.enable` | Enable the use of the MPI parcelport. HPX tries to detect if the application was started within a parallel MPI environment. If the detection was succesful, the MPI parcelport is enabled by default. To explicitly disable the MPI parcelport, set to 0. Note that the initial bootstrap of the overall *HPX* application will be performed using MPI as well. |

| **Property** | **Description** |
|---|---|
| `hpx.parcel.mpi.env` | This property influences which environment variables (comma separated) will be analyzed to find out whether the application was invoked by MPI. |
| `hpx.parcel.mpi.multithreaded` | This property is used to determine what threading mode to use when initializing MPI. If this setting is `0`, *HPX* will initialize MPI with `MPI_THREAD_SINGLE`, if the value is not equal to `0` *HPX* will will initialize MPI with `MPI_THREAD_MULTI`. |
| `hpx.parcel.mpi.rank` | This property will be initialized to the MPI rank of the locality. |
| `hpx.parcel.mpi.processor_name` | This property will be initialized to the MPI processor name of the locality. |
| `hpx.parcel.mpi.array_optimization` | This property defines whether this locality is allowed to utilize array optimizations in the MPI parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.array_optimization`. |
| `hpx.parcel.mpi.zero_copy_optimization` | This property defines whether this locality is allowed to utilize zero copy optimizations in the MPI parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`. |
| `hpx.parcel.mpi.use_io_pool` | This property can be set to run the progress thread inside of HPX threads instead of a separate thread pool. The default is `1`. |
| `hpx.parcel.mpi.async_serialization` | This property defines whether this locality is allowed to spawn a new thread for serialization in the MPI parcelport (this is both for encoding and decoding parcels). The default is the same value as set for `hpx.parcel.async_serialization`. |
| `hpx.parcel.mpi.enable_security` | This property defines whether this locality is encrypting parcels in the MPI parcelport. The default is the same value as set for `hpx.parcel.enable_security`. |

#### *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_initial_agas_local_cache_size>}
local_cache_size_per_thread = -
${HPX_AGAS_LOCAL_CACHE_SIZE_PER_THREAD:<hpx_initial_agas_local_cache_size_per_thread>}
```

| Property | Description |
|---|---|
| hpx.agas.address | This property defines the default IP address to be used for the AGAS root server. This IP address will be used as long as no other values are specified (for instance using the --hpx:agas command line option). The expected format is any valid IP address or domain name format which can be resolved into an IP address. The default depends on the compile time preprocessor constant HPX_INITIAL_IP_ADDRESS ("127.0.0.1"). |
| hpx.agas.port | This property defines the default IP port to be used for the AGAS root server. This IP port will be used as long as no other values are specified (for instance using the --hpx:agas command line option). The default depends on the compile time preprocessor constant HPX_INITIAL_IP_PORT (7010). |
| hpx.agas.service_mode | This property specifies what type of AGAS service is running on this locality. Currently, two modes exist. The locality that acts as the AGAS server runs in bootstrap mode. All other localities are in hosted mode. |
| hpx.agas.dedicated_server | This property specifies whether the AGAS server is exclusively running AGAS services and not hosting any application components. It is a boolean value. Set to 1 if --hpx-run-agas-server-only is present. |
| hpx.agas.max_pending_refcnt_requests | This property defines the number of reference counting requests (increments or decrements) to buffer. The default depends on the compile time preprocessor constant HPX_INITIAL_AGAS_MAX_PENDING_REFCNT_REQUESTS (4096). |
| hpx.agas.use_caching | This property specifies whether a software address translation cache is used. It is a boolean value. Defaults to 1. |
| hpx.agas.use_range_caching | This property specifies whether range-based caching is used by the software address translation cache. This property is ignored if hpx.agas.use_caching is false. It is a boolean value. Defaults to 1. |
| hpx.agas.local_cache_size | This property defines the size of the software address translation cache for AGAS services. This property is ignored if hpx.agas.use_caching is false. Note that if hpx.agas.use_range_caching is true, this size will refer to the maximum number of ranges stored in the cache, not the number of entries spanned by the cache. Note also, that the size of the cache will be determined by the larger of the following two numbers: the value of this key and the value of hpx.agas.local_cache_size_per_thread multiplied by the number of threads used systemwide in the running application. The default depends on the compile time preprocessor constant HPX_INITIAL_AGAS_LOCAL_CACHE_SIZE (256). |

| Property | Description |
|---|---|
| `hpx.agas.local_cache_size_per_thread` | This property defines the size of the software address translation cache for AGAS services on a per node basis. This property is ignored if `hpx.agas.use_caching` is false. Note that if `hpx.agas.use_range_caching` is true, this size will refer to the maximum number of ranges stored in the cache, not the number of entries spanned by the cache. Note also, that the size of the cache will be determined by the larger of the following two numbers: the `hpx.agas.local_cache_size` and the value of this key multiplied by the number of threads used system wide in the running application. The default depends on the compile time preprocessor constant `HPX_AGAS_LOCAL_CACHE_SIZE_PER_THREAD` (32). |

### *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]
-a = ---hpx:agas
-c = ---hpx:console
-h = ---hpx:help
--help = ---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
--version = ---hpx:version
-w = ---hpx:worker
-x = ---hpx:hpx
-0 = ---hpx:node=0
-1 = ---hpx:node=1
-2 = ---hpx:node=2
-3 = ---hpx:node=3
-4 = ---hpx:node=4
-5 = ---hpx:node=5
-6 = ---hpx:node=6
-7 = ---hpx:node=7
-8 = ---hpx:node=8
-9 = ---hpx:node=9
```

| Property | Description |
|---|---|
| `hpx.commandline.-a` | On the commandline, `-a` expands to: `--hpx:agas` |
| `hpx.commandline.-c` | On the commandline, `-c` expands to: `--hpx:console` |
| `hpx.commandline.-h` | On the commandline, `-h` expands to: `--hpx:help` |
| `hpx.commandline.--help` | On the commandline, `--help` expands to: `--hpx:help` |

| Property | Description |
|---|---|
| `hpx.commandline.-I` | On the commandline, `-I` expands to: `--hpx:ini` |
| `hpx.commandline.-l` | On the commandline, `-l` expands to: `--hpx:localities` |
| `hpx.commandline.-p` | On the commandline, `-p` expands to: `--hpx:app-config` |
| `hpx.commandline.-q` | On the commandline, `-q` expands to: `--hpx:queuing` |
| `hpx.commandline.-r` | On the commandline, `-r` expands to: `--hpx:run-agas-server` |
| `hpx.commandline.-t` | On the commandline, `-t` expands to: `--hpx:threads` |
| `hpx.commandline.-v` | On the commandline, `-v` expands to: `--hpx:version` |
| `hpx.commandline.--version` | On the commandline, `--version` expands to: `--hpx:version` |
| `hpx.commandline.-w` | On the commandline, `-w` expands to: `--hpx:worker` |
| `hpx.commandline.-x` | On the commandline, `-x` expands to: `--hpx:hpx` |
| `hpx.commandline.-0` | On the commandline, `-0` expands to: `--hpx:node=0` |
| `hpx.commandline.-1` | On the commandline, `-1` expands to: `--hpx:node=1` |
| `hpx.commandline.-2` | On the commandline, `-2` expands to: `--hpx:node=2` |
| `hpx.commandline.-3` | On the commandline, `-3` expands to: `--hpx:node=3` |
| `hpx.commandline.-4` | On the commandline, `-4` expands to: `--hpx:node=4` |
| `hpx.commandline.-5` | On the commandline, `-5` expands to: `--hpx:node=5` |
| `hpx.commandline.-6` | On the commandline, `-6` expands to: `--hpx:node=6` |
| `hpx.commandline.-7` | On the commandline, `-7` expands to: `--hpx:node=7` |
| `hpx.commandline.-8` | On the commandline, `-8` expands to: `--hpx:node=8` |
| `hpx.commandline.-9` | On the commandline, `-9` expands to: `--hpx:node=9` |

## Loading INI Files

During startup and after the internal database has been initialized as described in the section Built-in Default Configuration Settings, *HPX* will try to locate and load additional ini files to be used as a source for configuration properties. This allows for a wide spectrum of additional customization possibilities by the user and system administrators. The sequence of locations where *HPX* will try loading the ini files is well defined and documented in this section. All ini files found are merged into the internal configuration database. The merge operation itself conforms to the rules as described in the section The *HPX* INI File Format.

1. Load all component shared libraries found in the directories specified by the property `hpx.component_path` and retrieve their default configuration information (see section Loading Components for more details). This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows).

2. Load all files named `hpx.ini` in the directories referenced by the property `hpx.master_ini_path`. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows).
3. Load a file named `.hpx.ini` in the current working directory, e.g. the directory the application was invoked from.
4. Load a file referenced by the environment variable `HPX_INI`. This variable is expected to provide the full path name of the ini configuration file (if any).
5. Load a file named `/etc/hpx.ini`. This lookup is done on non-Windows systems only.
6. Load a file named `.hpx.ini` in the home directory of the current user, e.g. the directory referenced by the environment variable `HOME`.
7. Load a file named `.hpx.ini` in the directory referenced by the environment variable `PWD`.
8. Load the file specified on the command line using the option `--hpx:config`.
9. Load all properties specified on the command line using the option `--hpx:ini`. The properties will be added to the database in the same sequence as they are specified on the command line. The format for those options is for instance `--hpx:ini=hpx.default_stack_size=0x4000`. In adddition to the explicit command line options, this will set the following properties as implied from other settings:
   - `hpx.parcel.address` and `hpx.parcel.port` as set by `--hpx:hpx`
   - `hpx.agas.address`, `hpx.agas.port`, and `hpx.agas.service_mode` as set by `--hpx:agas`
   - `hpx.program_name` and `hpx.cmd_line` will be derived from the actual command line
   - `hpx.os_threads`, and `hpx.localities` as set by `--hpx:threads` and `--hpx:localities`
   - `hpx.runtime_mode` will be derived from any explicit `--hpx:console`, `--hpx:worker`, or `--hpx:connect`, or it will be derived from other settings, such as `--hpx:node=0` which implies `--hpx:console`
10. Load files based on the pattern `*.ini` in all directories listed by the property `hpx.ini_path`. All files found during this search will be merged. The property `hpx.ini_path` can hold a list of directories separated by `':'` (on Linux or Mac) or `';'` (on Windows)
11. Load the file specified on the command line using the option `--hpx:app-config`. Note that this file will be merged as the content for a top level section `[application]`.

## Note

Any changes made to the configuration database caused by one of the steps will influence the loading process for all subsequent steps. For instance, if one of the ini files loaded changes the property `hpx.ini_path`, this will influence the directories searched in step 9 as described above.

## Important

The *HPX* core library will verify that all configuration settings specified on the command line (using the `--hpx:ini` option) will be checked for validity. That means that the library will accept only *known* configuration settings. This is to protect the user from unintentional typos while specifying those settings. This behavior can be overwritten by appending a `'!'` to the configuration key, thus forcing the setting to be entered into the configuration database, for instance: `--hpx:ini=hpx.foo! = 1`.

If any of the environment variables or files listed above is not found the corresponding loading step will be silently skipped.

## Loading Components

*HPX* relies on loading application specific components during the runtime of an application. Moreover, *HPX* comes with a set of preinstalled components supporting basic functionalities useful for almost every application. Any component in *HPX* is loaded from a shared library, where any of the shared libraries can contain more than one component type. During startup, *HPX* tries to locate all available components (e.g. their corresponding shared libraries) and creates an internal component registry for later use. This section describes the algorithm used by *HPX* to locate all relevant shared libraries on a system. As described, this algorithm is customizable by the configuration properties loaded from the ini files (see section Loading INI Files).

Loading components is a two stage process. First *HPX* tries to locate all component shared libraries, loads those, and generates default configuration section in the internal configuration database for each component found. For each found component the following information is generated:

```
[hpx.components.<component_instance_name>]
name = <name_of_shared_library>
path = $[component_path]
enabled = $[hpx.components.load_external]
default = 1
```

The values in this section correspond to the expected configuration information for a component as described in the section Built-in Default Configuration Settings.

In order to locate component shared libraries, *HPX* will try loading all shared libraries (files with the platform specific extension of a shared library, Linux: `*.so`, Windows: `*.dll`, MacOS: `*.dylib`) found in the directory referenced by the ini property `hpx.component_path`.

This first step corresponds to step 1) during the process of filling the internal configuration database with default information as described in section Loading INI Files.

After all of the configuration information has been loaded, *HPX* performs the second step in terms of loading components. During this step, *HPX* scans all existing configuration sections `[hpx.component.<some_component_instance_name>]` and instantiates a special factory object for each of the successfully located and loaded components. During the application's life time, these factory objects will be responsible to create new and discard old instances of the component they are associated with. This step is performed after step 11) of the process of filling the internal configuration database with default information as described in section Loading INI Files.

### Application Specific Component Example

In this section we assume to have a simple application component which exposes one member function as a component action. The header file `app_server.hpp` declares the C++ type to be exposed as a component. This type has a member function `print_greating()` which is exposed as an action (`print_greating_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_greating'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::simple_component_base<server>
    {
        void print_greating ()
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }
        // Component actions need to be declared, this also defines the
        // type -'print_greating_action' representing the action.
        HPX_DEFINE_COMPONENT_ACTION(server, print_greating, print_greating_action);
    };
}

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

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

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

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

// Define factory object associated with our component of type -'app::server'.
HPX_REGISTER_MINIMAL_COMPONENT_FACTORY(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_greating_action);
```

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

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

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

        // Create an instance of the action -'print_greating_action'.
        app::server::print_greating_action print_greating;

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

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

In order to make sure that the application will be able to use the component `app::server`, special configuration information must be passed to *HPX*. The simples way to allow *HPX* to 'find' the component is to provide special ini configuration files, which add the necessary information to the internal configuration database. The component should have a special ini file containing the information specific to the component `app_server`:

```
# file: $APP_ROOT/app_server.ini
[hpx.components.app_server]
name = app_server
path = $APP_LOCATION/
```

Here `$APP_LOCATION` is the directory where the (binary) component shared library is located. *HPX* will attempt to load the shared library from there. The section name `hpx.components.app_server` reflects the instance name of the component (`app_server` is an arbitrary, but unique name) . The property value for `hpx.components.app_server.name` should be the same as used for the second argument to the macro `HPX_REGISTER_MINIMAL_COMPONENT_FACTORY` above.

Additionally a file `.hpx.ini` which could be located in the current working directory (see step 3 as described in the section Loading INI Files) can be used to add to the ini search path for components:

```
# file: $PWD/.hpx.ini
[hpx]
ini_path = $[hpx.ini_path]:$APP_ROOT/
```

This assumes that the above ini file specific to the component is located in the directory `$APP_ROOT`.

**Note**

It is possible to reference the defined property from inside its value. *HPX* will gracefully use the previous value of `hpx.ini_path` for the reference on the right hand side and assign the overall (now expanded) value to the property.

## Logging

*HPX* uses a sophisticated logging framework allowing to follow in detail what operations have been performed inside the *HPX* library in what sequence. This information proves to be very useful for diagnosing problems or just for improving the understanding what is happening in *HPX* as a consequence of invoking *HPX* API functionality.

### Default Logging

Enabling default logging is a simple process. The detailed description in the remainder of this section explains different ways to customize the defaults. Default logging can be enabled by using one of the following:

- a command line switch `--hpx:debug-hpx-log`, which will enable logging to the console terminal
- the command line switch `--hpx:debug-hpx-log=<filename>`, which enables logging to a given file `<filename>`, or
- setting an environment variable `HPX_LOGLEVEL=<loglevel>` while running the *HPX* application. In this case `<loglevel>` should be a number between (or equal to) `1` and `5`, where `1` means minimal logging and `5` causes to log all available messages. When setting a the environment variable the logs will be written to a file named `hpx.<PID>.login` the current working directory, where `<PID>` is the process id of the console instance of the application.

### Customizing Logging

Generally, logging can be customized either using environment variable settings or using by an ini configuration file. Logging is generated in several categories, each of which can be customized independently. All customizable configuration parameters have reasonable defaults, allowing to use logging without any additional configuration effort. The following table lists the available categories.

### Table 9. Logging categories

| Category | Category shortcut | Information to be generated | Environment variable |
|---|---|---|---|
| General | None | Logging information generated by different subsystems of *HPX*, such as thread-manager, parcel layer, LCOs, etc. | `HPX_LOGLEVEL` |
| AGAS | `AGAS` | Logging output generated by the AGAS subsystem | `HPX_AGAS_LOGLEVEL` |
| Application | `APP` | Logging generated by applications. | `HPX_APP_LOGLEVEL` |

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

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

## Levels

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

### Table 10. Logging levels

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

**Tip**

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

## Configuration

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

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

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

### Table 11. Logging destinations

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

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

### Table 12. Available field placeholders

| Name | Description |
|---|---|
| locality | The id of the locality on which the logging message was generated. |
| hpxthread | The id of the *HPX*-thread generating this logging output. |
| hpxphase | The phase[a] of the *HPX*-thread generating this logging output. |

| Name | Description |
|---|---|
| hpxcomponent | The local virtual address of the component which the current *HPX*-thread is accessing. |
| parentloc | The id of the locality where the *HPX* thread was running which initiated the current *HPX*-thread. The current *HPX*-thread is generating this logging output. |
| hpxparent | The id of the *HPX*-thread which initiated the current *HPX*-thread. The current *HPX*-thread is generating this logging output. |
| hpxparentphase | The phase of the *HPX*-thread when it initiated the current *HPX*-thread. The current *HPX*-thread is generating this logging output. |
| time | The time stamp for this logging outputline as generated by the source locality. |
| idx | The sequence number of the logging output line as generated on the source locality. |
| osthread | The sequence number of the OS-thread which executes the current *HPX*-thread. |

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

## Note

Not all of the field placeholder may be expanded for all generated logging output. If no value is available for a particular field it is replaced with a sequence of `'-'` characters.

Here is an example line from a logging output generated by one of the *HPX* examples (please note that this is generated on a single line, without line break):

```
(T00000000/0000000002d46f90.01/00000000009ebc10) P--------/0000000002d46f80.02 17:49.37.320 -
[000000000000004d]
    <info>  [RT] successfully created component {0000000100ff0001, 0000000000030002} of type: -
component_barrier[7(3)]
```

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

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

These settings define how the logging is customized once the logging output is received by the console instance of an application. The logging level is read from the environment variable `HPX_LOGLEVEL` (as set for the console instance of the application). The level defaults to the same values as the corresponding settings in the general logging configuration shown before. The destination on the console instance is set to be a file which name is generated based from its OS process id. Setting the environment variable `HPX_CONSOLE_LOGDESTINATION` allows customization of the naming scheme for the output file. The logging format is set to leave the original logging output unchanged, as received from one of the localities the application runs on.

## *HPX* Command Line Options

The predefined command line options for any application using `hpx::init` are described in the table below:

### Table 13. Default HPX Command Line Options

| Option | Description |
|---|---|
| **HPX options (allowed on command line only)** | |
| `--hpx:help`, `--help` or `-h` | print out program usage (default: this message), possible values: 'full' (additionally prints options from components) |
| `--hpx:version`, `--version` or `-v` | print out *HPX* version and copyright information |
| `--hpx:info` | print out *HPX* configuration information |
| `--hpx:options-file arg` | specify a file containing command line options (alternatively: @filepath) |
| **HPX options (additionally allowed in an options file)** | |
| `--hpx:worker` | run this instance in worker mode |
| `--hpx:console` | run this instance in console mode |
| `--hpx:connect` | run this instance in worker mode, but connecting late |
| `--hpx:run-agas-server` | run AGAS server as part of this runtime instance |
| `--hpx:run-hpx-main` | run the hpx_main function, regardless of locality mode |
| `--hpx:hpx arg` | the IP address the *HPX* parcelport is listening on, expected format: 'address:port' (default: 127.0.0.1:7910) |
| `--hpx:agas arg` | the IP address the AGAS root server is running on, expected format: 'address:port' (default: 127.0.0.1:7910) |
| `--hpx:run-agas-server-only` | run only the AGAS server |
| `--hpx:nodefile arg` | the file name of a node file to use (list of nodes, one node name per line and core) |
| `--hpx:nodes arg` | the (space separated) list of the nodes to use (usually this is extracted from a node file) |
| `--hpx:endnodes` | this can be used to end the list of nodes specified using the option `--hpx:nodes` |
| `--hpx:ifsuffix arg` | suffix to append to host names in order to resolve them to the proper network interconnect |

| Option | Description |
|---|---|
| `--hpx:ifprefix arg` | prefix to prepend to host names in order to resolve them to the proper network interconnect |
| `--hpx:iftransform arg` | sed-style search and replace (s/search/replace/) used to transform host names to the proper network interconnect |
| `--hpx:localities arg` | the number of localities to wait for at application startup (default: 1) |
| `--hpx:node arg` | number of the node this locality is run on (must be unique) |
| `--hpx:pu-offset` | the first processing unit this instance of *HPX* should be run on (default: 0), valid for --hpx:queuing=local and priority_local only |
| `--hpx:pu-step` | the step between used processing unit numbers for this instance of *HPX* (default: 1), valid for `--hpx:queuing=local` and priority_local only |
| `--hpx:threads arg` | the number of operating system threads to spawn for this *HPX* locality (default: 1, using 'all' will spawn one thread for each processing unit) |
| `--hpx:cores arg` | the number of cores to utilize for this *HPX* locality 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), valid for `--hpx:queuing=local` and priority_local only |
| `--hpx:bind arg` | the detailed affinity description for the OS threads, see the additional documentation 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: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/l', 'priority_local/pr', 'abp/a', 'priority_abp', 'hierarchy/h', and 'periodic/pe' (default: priority_local/p) |
| `--hpx:hierarchy-arity` | the arity of the of the thread queue tree, valid for --hpx:queuing=hierarchy only (default: 2) |
| `--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=priority_local and --hpx:queuing=priority_abp only |

| **Option** | **Description** |
|---|---|
| `--hpx:numa-sensitive` | makes the priority_local scheduler NUMA sensitive, valid for `--hpx:queuing=local` and priority_local only |
| **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` | enable all messages on the *HPX* log channel and send all *HPX* logs to the target destination |
| `--hpx:debug-agas-log` | enable all messages on the AGAS log channel and send all AGAS logs to the target destination |
| `--hpx:debug-clp` | debug command line processing |
| **HPX options related to performance counters** | |
| `--hpx:print-counter` | print the specified performance counter either repeatedly or before shutting down the system (see 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) |

## Command Line Argument Shortcuts

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

### Table 14. Predefined command line option shortcuts

| Shortcut option | Equivalent long option |
|---|---|
| `-a` | `--hpx:agas` |
| `-c` | `--hpx:console` |
| `-h` | `--hpx:help` |
| `--help` | `--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` |
| `--version` | `--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` |

| Shortcut option | Equivalent long option |
|---|---|
| `-9` | `--hpx:node=9` |

It is possible to define your own shortcut options. In fact, all of the shortcuts listed above are pre-defined using the technique described here. In fact, 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 Configure *HPX* Applications. For instance, in order to define a command line shortcut `--pc` which should expand to `--hpx:print-counter`, the following configuration information needs to be added to one of the `ini` configuration files:

```
[hpx.commandline]
--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 Using PBS), specifying dedicated command line options for single localities may be desirable. For this reason all of the command line options which have the general format `--hpx:<some_key>` can be used in a more general form: `--hpx:<N>:<some_key>`, where <N> is the number of the locality this command line options will be applied to, all other localities will simply ignore the option. For instance, the following PBS script passes the option `--hpx:pu-offset=4` to the locality `'1'` only.

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

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

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

**Caution**

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

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

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

## More Details about *HPX* Command Line Options

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

- `--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.

**Note**

This command line option is only available if *HPX* was built with support for HWLOC (Portable Hardware Locality (HWLOC)) enabled. Please see Options and Variables for more details on how to enable support for HWLOC in *HPX*.

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

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

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

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

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

**Note**

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

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

```
mappings:
    distribution
    mapping(;mapping)*

distribution:
    -'compact'
    -'scatter
    -'balanced'

mapping:
    thread-spec=pu-specs

thread-spec:
    -'thread':range-specs

pu-specs:
    pu-spec(.pu-spec)*

pu-spec:
    type:range-specs
    ~pu-spec

range-specs:
    range-spec(,range-spec)*

range-spec:
    int
    int-int
    -'all'

type:
    -'socket' -| -'numanode'
    -'core'
    -'pu'
```

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

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

Here `thread:0-3` specifies the OS threads for which to define affinity bindings, and `core:0-3.pu: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 --t4 ---hpx:bind=thread:0-3=core:0-3.pu:0 ---hpx:print-bind
```

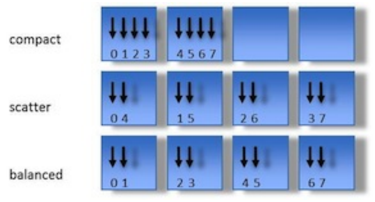
will cause this output to be printed:

```
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 three possible predefined distribution schemes (`compact`, `scatter`, and `balanced`) is best explained with an example. Imagine that we have a system with 4 cores and 4 hardware threads per core. If we place 8 threads the assignments produced by the `compact`, `scatter`, and `balanced` types are shown in eh 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` options in most cases.

**Figure 2. Schematic of thread affinity type distributions**



# Writing *HPX* applications

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

- Minimally invasive: Re-use the `main()` function as the main *HPX* entry point.
- Balanced use case: Supply your own main *HPX* entry point while blocking the main thread.
- Most flexibility: Supply your own main *HPX* entry point while avoiding to block the main thread.

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

This method is the least intrusive to your code. It however provides you with the smallest flexibility in terms of initializing the *HPX* runtime system only. 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 stopped after `main()` has returned. All *HPX* API functions can be used from within this function now.

**Note**

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

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

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.

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

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

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

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

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

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

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

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

**Note**

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

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

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

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

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

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

The thread executing the function `hpx::start` will **not** block waiting for the runtime system to exit, but will return immediatlely.

**Important**

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

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

This method of invoking *HPX* is useful for applications where the main thread is used for special operations, such a GUIs. The function `hpx::stop` can be used to wait for the *HPX* runtime system to exit and should be at least used as the last function called in `main()`. The value returned from `hpx_main` will be returned from `hpx::stop` after the runtime system has stopped.

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

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

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

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

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

**Note**

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

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

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

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

## Global Names

*HPX* implements an Active Global Address Space (AGAS) which is exposing a single uniform address space spanning all localities an application runs on. AGAS is a fundamental component of the ParalleX execution model. Conceptually, there is no rigid demarcation of local or global memory in AGAS; all available memory is a part of the same address space. AGAS enables named objects to be moved (migrated) across localities without having to change the object's name, i.e., no references to migrated objects have to be ever updated. This feature has significance for dynamic load balancing and in applications where the workflow is highly dynamic, allowing work to be migrated from heavily loaded nodes to less loaded nodes. In addition, immutability of names ensures that AGAS does not have to keep extra indirections ("bread crumbs") when objects move, hence minimizing complexity of code management for system developers as well as minimizing overheads in maintaining and managing aliases.

The AGAS implementation in *HPX* does not automatically expose every local address to the global address space. It is the responsibility of the programmer to explicitly define which of the objects have to be globally visible and which of the objects are purely local.

In *HPX* global addresses (global names) are represented using the `hpx::id_type` data type. This data type is conceptually very similar to `void*` pointers as it does not expose any type information of the object it is referring to.

The only predefined global addresses are assigned to all localities. The following *HPX* API functions allow one to retrieve the global addresses of localities:

- `hpx::find_here()`: retrieve the global address of the locality this function is called on.
- `hpx::find_all_localities()`: retrieve the global addresses of all localities available to this application (including the locality the function is being called on).
- `hpx::find_remote_localities()`: retrieve the global addresses of all remote localities available to this application (not including the locality the function is being called on)
- `hpx::get_num_localities()`: retrieve the number of localities available to this application.
- `hpx::find_locality()`: retrieve the global address of any locality supporting the given component type.
- `hpx::get_colocation_id()`: retrieve the global address of the locality currently hosting the object with the given global address.

Additionally, the global addresses of localities can be used to create new instances of components using the following *HPX* API function:

- `hpx::new_<Component>()`: Create a new instance of the given `Component` type on the specified locality.

**Note**

*HPX* does not expose any functionality to delete component instances. All global addresses (as represented using `hpx::id_type`) are being automatically garbage collected. When the last (global) reference to a particular component instance goes out of scope the corresponding component instance is automatically deleted.

## Applying Actions

### Action Type Definition

Actions are special types we use to describe possibly remote operations. For every global function and every member function which has to be invoked distantly, a special type must be defined. For any global function the special macro `HPX_PLAIN_ACTION` can be used to define the action type. Here is an example demonstrating this:

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }
}

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

**Important**

The macro `HPX_PLAIN_ACTION` has to be placed in global namespace, even if the wrapped function is located in some other namespace. The newly defined action type is placed the in 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_PLAIN_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_PLAIN_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`).

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::simple_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
        // obect type -'some_component'.
        HPX_DEFINE_COMPONENT_ACTION(some_component, some_member_function,
            some_member_action);
    };
}

// Note: The second arguments to the macro below have to be systemwide-unique
//       C++ identifiers
HPX_REGISTER_ACTION_DECLARATION(app::some_component::some_member_action , some_component_some_action );
```

The next snippet belongs to the source file (e.g. the main application in the most simple case:

```
typedef hpx::components::simple_component< app::some_component > component_type;
typedef app::some_component some_component;

HPX_REGISTER_MINIMAL_COMPONENT_FACTORY( component_type, some_component );

// The parameters for this macro have to be the same as used in the corresponding
// HPX_REGISTER_ACTION_DECLARATION() macro invocation above
typedef some_component::some_member_action some_component_some_action;
HPX_REGISTER_ACTION( some_component_some_action );
```

Granted, these macro invocations are a bit more complex than for simple global functions, however we believe they are still manageable.

The most important macro invocation is the `HPX_DEFINE_COMPONENT_ACTION` in the header file as this defines the action type we need to invoke the member function. For a complete example of a simple component action see component_in_executable.cpp

## Action Invocation

The process of invoking a global function (or a member function of an object) with the help of the associated action is called 'applying the action'. Actions can have arguments, which will be supplied while the action is applied. At the minimum, one parameter is required to apply any action - the id of the locality the associated function should be invoked on (for global functions), or the id of the component instance (for member functions). Generally, *HPX* provides several ways to apply an action, all of which are described in the following sections.

*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.

## 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 asyncronous 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 destriuctor, 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 asyncronous operation to -'return' 42
```

**Note**

The invocation of `f.get()` will return the result immediately (without suspending the calling thread) if the result from the asynchronous operation has already been returned. Otherwise, the invocation of `f.get()` will suspend the execution of the calling thread until the asynchronous operation returns its result.

## Applying an Action Synchronously

This method will schedule the function wrapped in the specified action on the target locality. While the invocation appears to be synchronous (as we will see), the calling thread will be suspended while waiting for the function to return. Invoking a plain action (e.g. a global function) synchronously is straightforward:

```
some_global_action act;     // define an instance of some_global_action
act(hpx::find_here(), 2.0);
```

While this call looks just like a normal synchronous function invocation, the function wrapped by the action will be scheduled to run on a new thread and the calling thread will be suspended. After the new thread has executed the wrapped global function, the waiting thread will resume and return from the synchronous call.

Equivalently, any action wrapping a component member function can be invoked synchronously as follows:

```
some_component_action act;     // define an instance of some_component_action
int result = act(id, "42");
```

The action invocation will either schedule a new thread locally to execute the wrapped member function (as before, `id` is the global address of the component instance the member function should be invoked on), or it will send a parcel to the remote locality of the component causing a new thread to be scheduled there. The calling thread will be suspended until the function returns its result. This result will be returned from the synchronous action invocation.

It is very important to understand that this 'synchronous' invocation syntax in fact conceals an asynchronous function call. This is beneficial as the calling thread is suspended while waiting for the outcome of a potentially remote operation. The *HPX* thread scheduler will schedule other work in the mean time, allowing the application to make further progress while the remote result is computed. This helps overlapping computation with communication and hiding communication latencies.

**Note**

The syntax of applying an action is always the same, regardless whether the target locality is remote to the invocation locality or not. This is a very important feature of *HPX* as it frees the user from the task of keeping track what actions have to be applied locally and which actions are remote. If the target for applying an action is local, a new thread is automatically created and scheduled. Once this thread is scheduled and run, it will execute the function encapsulated by that action. If the target is remote, *HPX* will send a parcel to the remote locality which encapsulates the action and its parameters. Once the parcel is received on the remote locality *HPX* will create and schedule a new thread there. Once this thread runs on the remote locality, it will execute the function encapsulated by the action.

## Applying an Action with a Continuation but without any Synchronization

This method is very similar to the method described in section Applying an Action Asynchronously without any Synchronization. The difference is that it allows the user to chain a sequence of asynchronous operations, while handing the (intermediate) results from one step to the next step in the chain. Where `hpx::apply` invokes a single function using 'fire and forget' semantics, `hpx::apply_continue` asynchronously triggers a chain of functions without the need for the execution flow 'to come back' to the invocation site. Each of the asynchronous functions can be executed on a different locality.

## Applying an Action with a Continuation and with Synchronization

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

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

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

// second action
boost::int32_t action2(boost::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::find_here(), 42,
        hpx::make_continuation(act2));
hpx::cout << f.get() << "\n";   // will print: 86 ((42 + 1) * 2)
```

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

```
// this code invokes -'action1' above and passes along a continuation
// function which will forward the result returned from -'action1' to
// -'action2'.
action1_type act1;     // define an instance of -'action1_type'
```

```
action2_type act2;     // define an instance of -'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1, hpx::find_here(), 42,
        hpx::make_continuation(act2, hpx::find_here()));
hpx::cout << f.get() << "\n";   // will print: 86 ((42 + 1) * 2)
```

Similarily, 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::find_here(), 42,
        hpx::make_continuation(act2,
            hpx::make_continuation(act1)));
hpx::cout << f.get() << "\n";   // will print: 87 ((42 + 1) * 2 + 1)
```

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

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

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

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

**Note**

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

## Comparing Actions with C++ Functions

As mentioned above, *HPX* actions are very similar to 'normal' C++ functions except that actions can be invoked remotely. The following table compares the way C++ functions can be invoked with the syntax used to apply an *HPX* action. This table assumes that `func` is an arbitrary C++ function or function object and `act` is an instance of an *HPX* action. `R` stands for an arbitrary result type.

### Table 15. Comparing C++ Functions and *HPX* Actions

| Invocation | C++ Function | *HPX* Action |
|---|---|---|
| Apply (fire & forget) | `hpx::apply(func, ...);` | `hpx::apply(act, id, ...);` |

| **Invocation** | **C++ Function** | ***HPX* Action** |
|---|---|---|
| Asynchronous | `std::future<R> f = std::async(func, ...);`<br>`R r = f.get();`<br><br>`hpx::future<R> f = hpx::async(func, ...);`<br>`R r = f.get();` | `hpx::future<R> f = hpx::async(act, id,`<br>`R r = f.get();` |
| Synchronous | `R r = f(...)` | `R r = act(id, ...)` |

## 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 Error Handling 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_argument, "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 Error Handling. There we also explain how to handle error conditions without having to rely on exception.

## Writing Plain Actions

## Writing Components

### Component Server Classes

### Component Stubs Classes

### Component Client Classes

## Using LCOs

Lightweight Control Objects provide synchrhonization for HPX applications. Most of them are familiar from other frameworks, but a few of them work in slightly special different ways adapted to HPX.

1. future
2. queue
3. object_semaphore
4. barrier
5. and_gate
6. barrier
7. channel

8. composable_guard - Composable guards operate in a manner similar to locks, but are applied only to asynchronous functions. The guard (or guards) is automatically locked at the beginning of a specified task and automatically unlocked at the end. Because guards are never added to an existing task's execution context, the calling of guards is freely composable and can never deadlock.

To call an application with a single guard, simply declare the guard and call run_guarded() with a function (task).

```
hpx::lcos::local::guard gu;
run_guarded(gu,task);
```

If a single method needs to run with multiple guards, use a guard set.

```
boost::shared<hpx::lcos::local::guard> gu1(new hpx::lcos::local::guard());
boost::shared<hpx::lcos::local::guard> gu2(new hpx::lcos::local::guard());
gs.add(*gu1);
gs.add(*gu2);
run_guarded(gs,task);
```

Guards use two atomic operations (which are not called repeatedly) to manage what they do, so overhead should be extremely low.

1. conditional_trigger
2. counting_semaphore
3. dataflow
4. event
5. mutex
6. once
7. recursive_mutex
8. spinlock
9. spinlock_no_backoff

trigger

# Error Handling

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

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

## Working with Exceptions

For the following description we assume that the function `raise_exception()` is executed by invoking the plain action `raise_exception_type`:

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

The exception is thrown using the macro `HPX_THROW_EXCEPTION`. The type of the thrown exception is `hpx::exception`. This associates additional diagnostic information with the exception, such as file name and line number, locality id and thread id, and stack backtrace from the point where the exception was thrown.

Any exception thrown during the execution of an action is transferred back to the (asynchronous) invocation site. It will be rethrown in this context when the calling thread tries to wait for the result of the action by invoking either `future<>::get()` or the synchronous action invocation wrapper as shown here:

```
hpx::cout << "Error reporting using exceptions\n";
try {
    // invoke raise_exception() which throws an exception
    raise_exception_action do_it;
    do_it(hpx::find_here());
}
catch (hpx::exception const& e) {
    // Print just the essential error information.
    hpx::cout << "caught exception: -" << e.what() << "\n\n";

    // Print all of the available diagnostic information as stored with
    // the exception.
    hpx::cout << "diagnostic information:"
        << hpx::diagnostic_information(e) << "\n";
}
hpx::cout << hpx::flush;
```

**Note**

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

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

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

```
hpx::cout << "Detailed error reporting using exceptions\n";
try {
    // Invoke raise_exception() which throws an exception.
    raise_exception_action do_it;
    do_it(hpx::find_here());
}
catch (hpx::exception const& e) {
    // Print the elements of the diagnostic information separately.
    hpx::cout << "{what}: -"        << hpx::get_error_what(e) << "\n";
    hpx::cout << "{locality-id}: -" << hpx::get_error_locality_id(e) << "\n";
    hpx::cout << "{hostname}: -"    << hpx::get_error_host_name(e) << "\n";
    hpx::cout << "{pid}: -"         << hpx::get_error_process_id(e) << "\n";
    hpx::cout << "{function}: -"    << hpx::get_error_function_name(e) << "\n";
    hpx::cout << "{file}: -"        << hpx::get_error_file_name(e) << "\n";
    hpx::cout << "{line}: -"        << hpx::get_error_line_number(e) << "\n";
    hpx::cout << "{os-thread}: -"   << hpx::get_error_os_thread(e) << "\n";
    hpx::cout << "{thread-id}: -"   << std::hex << hpx::get_error_thread_id(e) << "\n";
    hpx::cout << "{thread-description}: -"
```

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

## Working with Error Codes

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

```
hpx::cout << "Error reporting using error code\n";

// Create a new error_code instance.
hpx::error_code ec;

// If an instance of an error_code is passed as the last argument while
// invoking the action, the function will not throw in case of an error
// but store the error information in this error_code instance instead.
raise_exception_action do_it;
do_it(hpx::find_here(), ec);

if (ec) {
    // Print just the essential error information.
    hpx::cout << "returned error: -" << ec.get_message() << "\n";

    // Print all of the available diagnostic information as stored with
    // the exception.
    hpx::cout << "diagnostic information:"
        << hpx::diagnostic_information(ec) << "\n";
}

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

**Note**

The error information is transferred back to the invocation site even if it is executed on a different locality.

This example show how an error can be handled without having to resolve to exceptions and that the returned `hpx::error_code` instance can be used in a very similar way as the `hpx::exception` type above. Simply pass it to the `hpx::diagnostic_information()` which retrieves all available diagnostic information from the error code instance as a formatted string.

As for handling exceptions, when working with error codes, under certain circumstances it is desireable to output only some of the diagnostics, or to output those using different formatting. For this case, *HPX* exposes a set of lower level functions usable with error codes as demonstrated in the following code snippet:

```
hpx::cout << "Detailed error reporting using error code\n";

// Create a new error_code instance.
hpx::error_code ec;
```

```
// If an instance of an error_code is passed as the last argument while
// invoking the action, the function will not throw in case of an error
// but store the error information in this error_code instance instead.
raise_exception_action do_it;
do_it(hpx::find_here(), ec);

if (ec) {
    // Print the elements of the diagnostic information separately.
    hpx::cout << "{what}: -"          << hpx::get_error_what(ec) << "\n";
    hpx::cout << "{locality-id}: -" << hpx::get_error_locality_id(ec) << "\n";
    hpx::cout << "{hostname}: -"    << hpx::get_error_host_name(ec) << "\n";
    hpx::cout << "{pid}: -"         << hpx::get_error_process_id(ec) << "\n";
    hpx::cout << "{function}: -"    << hpx::get_error_function_name(ec) << "\n";
    hpx::cout << "{file}: -"        << hpx::get_error_file_name(ec) << "\n";
    hpx::cout << "{line}: -"        << hpx::get_error_line_number(ec) << "\n";
    hpx::cout << "{os-thread}: -"   << hpx::get_error_os_thread(ec) << "\n";
    hpx::cout << "{thread-id}: -"   << std::hex << hpx::get_error_thread_id(ec) << "\n";
    hpx::cout << "{thread-description}: -"
        << hpx::get_error_thread_description(ec) << "\n\n";
    hpx::cout << "{stack-trace}: -" << hpx::get_error_backtrace(ec) << "\n";
    hpx::cout << "{env}: -"         << hpx::get_error_env(ec) << "\n";
}

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

For more information please refer to the documentation of `hpx::get_error_what()`, `hpx::get_error_locality_id()`, `hpx::get_error_host_name()`, `hpx::get_error_process_id()`, `hpx::get_error_function_name()`, `hpx::get_error_file_name()`, `hpx::get_error_line_number()`, `hpx::get_error_os_thread()`, `hpx::get_error_thread_id()`, and `hpx::get_error_thread_description()` `hpx::get_error_backtrace()` `hpx::get_error_env()` for more details.

## Lightweight error Codes

Sometimes it is not desireable to collect all the ambient information about the error at the point where it happened as this might impose too much overhead for simple scenarious. In this case, *HPX* provides a lightweight error code facility which will hold the error code only. The following snippet demonstrates its use:

```
hpx::cout << "Error reporting using an lightweight error code\n";

// Create a new error_code instance.
hpx::error_code ec(hpx::lightweight);

// If an instance of an error_code is passed as the last argument while
// invoking the action, the function will not throw in case of an error
// but store the error information in this error_code instance instead.
raise_exception_action do_it;
do_it(hpx::find_here(), ec);

if (ec) {
    // Print just the essential error information.
    hpx::cout << "returned error: -" << ec.get_message() << "\n";

    // Print all of the available diagnostic information as stored with
    // the exception.
    hpx::cout << "error code:" << ec.value() << "\n";
}
```

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

All functions which retrieve other diagnostic elements from the `hpx::error_code` will fail if called with a lightweight error_code instance.

## Performance Counters

Performance Counters in *HPX* are used to provide information as to how well the runtime system or an application is performing. The counter data can help determine system bottlenecks and fine-tune system and application performance. The *HPX* runtime system, its networking, and other layers provide counter data that an application can consume to provide users with information of how well the application is performing.

Applications can also use counter data to determine how much system resources to consume. For example, an application that transfers data over the network could consume counter data from a network switch to determine how much data to transfer without competing for network bandwidth with other network traffic. The application could use the counter data to adjust its transfer rate as the bandwidth usage from other network traffic increases or decreases.

Performance Counters are *HPX* parallel processes which expose a predefined interface. *HPX* exposes special API functions that allow one to create, manage, read the counter data, and release instances of Performance Counters. Performance Counter instances are accesssed 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 accesssed in real time. More information about how to consume counter data can be found in the section Consuming Performance Counters.

All *HPX* applications provide command line options related to performance counters, such as the ability to list available counter types, or periodically query specific counters to be printed to the screen or save them in a file. For more information, please refer to the section *HPX* Command Line Options.

### Performance Counter Names

All Performance Counter instances have a name uniquely identifying this instance. This name can be used to access the counter, retrieve all related meta data, and to query the counter data (as described in the section Consuming Performance Counters). Counter names are strings with a predefined structure. The general form of a countername is:

```
/objectname{full_instancename}/countername@parameters
```

where `full_instancename` could be either another (full) counter name or a string formatted as:

```
parentinstancename#parentindex/instancename#instanceindex
```

Each separate part of a countername (e.g. `objectname`, `countername`, `parentinstancename`, `instancename`, and `parameters`) should start with a letter (`'a'`...`'z'`, `'A'`...`'Z'`) or an underscore character (`'_'`), optionally followed by letters, digits (`'0'`...`'9'`), hyphen (`'-'`), or underscore characters. Whitespace is not allowed inside a counter name. The characters `'/'`, `'{'`, `'}'`, `'#'`, and `'@'` have a special meaning and are used to delimit the different parts of the counter name.

The parts `parentinstanceindex` and `instanceindex` are integers. If an index is not specified *HPX* will assume a default of `-1`.

#### Two Simple Examples

An instance for a well formed (and meaningful) simple counter name would be:

```
/threads{locality#0/total}/count/cumulative
```

This counter returns the current cumulative number of executed (retired) *HPX*-threads for the locality `0`. The counter type of this counter is `/threads/count/cumulative` and the full instance name is `locality#0/total` (**highlighted** for readability). This counter type does not require an `instanceindex` or `parameters` to be specified.

In this case, the `parentindex` (the `'0'`) designates the locality for which the counter instance is created. The counter will return the number of *HPX*-threads retired on that particular locality.

Another example for a well formed (aggregate) counter name is:

```
/statistics{/threads{locality#0/total}/count/cumulative}/average@500
```

This counter takes the simple counter from the first example, samples its values every `500` milliseconds, and returns the average of the value samples whenever it is queried. The counter type of this counter is `/statistics/average` and the instance name is the full name of the counter for which the values have to be averaged. In this case, the `parameters` (the `'500'`) specify the sampling interval for the averaging to take place (in milliseconds).

## Performance Counter Types

Every Performance Counter belongs to a specific Performance Counter type which classifies the counters into groups of common semantics. The type of a counter is identified by the `objectname` and the `countername` parts of the name.

```
/objectname/countername
```

At application start, *HPX* will register all available counter types on each of the localities. These counter types are held in a special Performance Counter registration database which can be later used to retrieve the meta data related to a counter type and to create counter instances based on a given counter instance name.

## Performance Counter Instances

The `full_instancename` distinguishes different counter instances of the same counter type. The formatting of the `full_instancename` depends on the counter type. There are two types of counters: simple counters which usually generate the counter values based on direct measurements, and aggregate counters which take another counter and transform its values before generating their own counter values. An example for a simple counter is given above: counting retired *HPX*-threads. An aggreagate counter is shown as an example above as well: calculating the average of the underlying counter values sampled at constant time intervals.

While simple counters use instance names formatted as `parentinstancename#parentindex/instancename#instanceindex`, most aggregate counters have the full counter name of the embedded counter as its instance name.

Not all simple counter types require specifying all 4 elements of a full counter instance name, some of the parts (`parentinstancename`, `parentindex`, `instancename`, and `instanceindex`) are optional for specific counters. Please refer to the documentation of a particular counter for more information about the formatting requirements for the name of this counter (see Existing 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 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 determinded dynamically at application start based on the execution environment of the application. For instance, this set is influenced by the current hardware environment for the locality (such as whether the locality has access to accelerators), and the software environment of the application (such as the number of OS-threads used to execute *HPX*-threads).

## Using Wildcards in Performance Counter Names

It is possible to use wildcard characters when specifying performance counter names. Performance counter names can contain 2 types of wildcard characters:

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

Wildcard character have a meaning which is very close to usual file name wildcard matching rules implemented by common shells (like bash).

### Table 16. Wildcard characters in the performance counter type

| **Wildcard** | **Description** |
|---|---|
| `*` | This wildchard character matches any number (zero or more) of arbitrary characters. |
| `?` | This wildchard character matches any single arbitrary character. |
| `[...]` | This wildchard character matches any single character from the list of specified within the square brackets. |

### Table 17. Wildcard characters in the performance counter instance name

| **Wildcard** | **Description** |
|---|---|
| `*` | This wildchard character matches any locality or any thread, depending on whether it is used for `locality#*` or `worker-thread#*`. No other wildcards are allowed in counter instance names. |

## Consuming Performance Counter Data

You can consume performance data using either the command line interface or via the *HPX* application or the *HPX* API. The command line interface is easier to use, but it is less flexible and does not allow one to adjust the behaviour of your application at runtime. The command line interface provides a convenience abstraction but simplified abstraction for querying and logging performance counter data for a set of performance counters.

### Consuming Performance Counter Data from the Command Line

*HPX* provides a set of predefined command line options for every application which uses `hpx::init` for its initialization. While there are much more command line options available (see *HPX* Command Line Options), the set of options related to Performance Counters allow one to list existing counters, query existing counters once at application termination or repeatedly after a constant time interval.

The following table summarizes the available command line options:

### Table 18. *HPX* Command Line Options Related to Performance Counters

| **Command line option** | **Description** |
|---|---|
| `--hpx:print-counter` | print the specified performance counter either repeatedly or before shutting down the system (see 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) |

| Command line option | Description |
|---|---|
| `--hpx:print-counter-destination` | print the performance counter(s) specified with `--hpx:print-counter` to the given file (default: console) |
| `--hpx:list-counters` | list the names of all registered performance counters |
| `--hpx:list-counter-infos` | list the description of all registered performance counters |

While the options `--hpx:list-counters` and `--hpx:list-counter-infos` give a short listing of all available counters, the full documentation for those can be found in the section Existing Performance Counters.

## A Simple Example

All of the commandline options mentioned above can be for instance tested using the `hello_world` example.

Listing all available counters (`hello_world --hpx:list-counters`) yields:

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

Providing more information about all available counters (`hello_world --hpx:list-counter-infos`) yields:

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

------------------------------------------------------------------------------
fullname: -/agas/count/bind
helptext: returns the number of invocations of the AGAS service -'bind'
type:     counter_raw
version:  1.0.0
------------------------------------------------------------------------------

------------------------------------------------------------------------------
fullname: -/agas/count/bind_gid
helptext: returns the number of invocations of the AGAS service -'bind_gid'
type:     counter_raw
version:  1.0.0
------------------------------------------------------------------------------
```

```
...
```

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

**Note**

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

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

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

which yields for instance:

```
hello world from OS-thread 0 on locality 0
/threads{locality#0/total}/count/cumulative,1,0.019633[s],36
/agas{root/total}/count/bind,1,0.020323[s],10
```

The first line is the normal output generated by hell_world and has no relation to the counter data listed. The last two lines contain the counter data as gathered at application shutdown. These lines have 4 fields, the counter name, the sequence number of the counter invocation, the time stamp at which this information has been sampled, and the actual counter value.

Requesting to query the counter data once after a constant time interval with this command line

```
hello_world \
    ---hpx:print-counter=/threads{locality#0/total}/count/cumulative \
    ---hpx:print-counter=/agas{root/total}/count/bind \
    ---hpx:print-counter-interval=20
```

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

```
threads{locality#0/total}/count/cumulative,1,0.002409[s],22
agas{root/total}/count/bind,1,0.002542[s],9
threads{locality#0/total}/count/cumulative,2,0.023002[s],41
agas{root/total}/count/bind,2,0.023557[s],10
threads{locality#0/total}/count/cumulative,3,0.037514[s],46
agas{root/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 occurences of `#*` as in `locality#*` and in `worker-thread#*` will be automatically expanded to the proper set of performance counter names representing the actual environment for the executed program. For instance, if your program is utilizing 4 worker threads for the execution of HPX threads (see command line option `--hpx:threads`) the following command line

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

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

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

### Consuming Performance Counter Data using the *HPX* API

## Providing Performance Counter Data

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

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

Generally, there two ways of exposing yur 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:

```
boost::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<boost::int64_t> counter(0);

boost::int64_t some_performance_data(bool reset)
{
    boost::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 incrememnted on each invocation, e.g. each time a consumer requests the counter data of this Performance Counter.

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

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

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

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

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

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

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

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

### Implementing a Full Performance Counter

Somtimes, 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:

```
// Abstract base interface for all Performance Counters.
struct performance_counter
{
    // Retrieve the descriptive information about the Performance Counter.
    virtual counter_info get_counter_info() const = 0;

    // Retrieve the current Performance Counter value.
    virtual counter_value get_counter_value(bool reset = false) = 0;

    // Reset the Performance Counter (value).
    virtual void reset_counter_value() = 0;

    // Set the (initial) value of the Performance Counter.
    virtual void set_counter_value(counter_value const& /*value*/) = 0;

    // Start the Performance Counter.
    virtual bool start() = 0;

    // Stop the Performance Counter.
    virtual bool stop() = 0;
};
```

In order to implement a full Performance Counter you have to create an *HPX* component exposing this interface. To simplify this task, *HPX* provides a ready made base class which handles all the boiler plate of creating a component for you. The remainder of this section will explain the process of creating a full Performance Counter based on the Sine example which you can find in the directory `examples/performance_counters/sine/`.

The base class is defined in the header file hpx/performance_counters/base_performance_counter.hpp as:

[base_performance_counter_class]

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:

```
class sine_counter
  : public hpx::performance_counters::base_performance_counter<sine_counter>
```

i.e. the type `sine_counter` is derived from the base class passing the type as a template argument (please see sine.hpp for the full source code of the counter definition). For more information about this technique (called Curiously Recurring Template Pattern - CRTP), please see for instance the corresponding Wikipedia article. This base class itself is derived from the `performance_counter` interface described above.

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

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

## Existing *HPX* Performance Counters

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

### Table 19. AGAS Performance Counters

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

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | AGAS services are available on `locality#0` only). | | |
| `/agas/time/<agas_service>`<br><br>where:<br>`<agas_service>` is one of the following: `allocate`, `bind`, `bind_gid`, `bind_name`, `bind_prefix`, `increment_credit`, `decrement_credit`, `free`, `get_component_typename`, `iterate_names`, `iterate_types`, `localities`, `num_localities`, `num_localities_type`, `num_threads`, `resolve`, `resolve_gid`, `resolve_id`, `resolve_locality`, `resolved_localities`, `route`, `unbind`, `unbind_gid`, `unbind_name`. | `<agas_instance>/total`<br><br>where:<br>`<agas_instance>` is the name of the AGAS service to query. Currently, this value will be `locality#0`, where `0` is the root locality (the id of the locality hosting the AGAS service). Except for `<agas_service>` `route`, `bind_gid`, `resolve_gid`, `unbind_gid`, `increment_credit`, `decrement_credit`,`bind`, `resolve`, `unbind`, and `iterate_names` for which the value for `*` can be any locality id (only the primary and symbol AGAS service components live on all localities, whereas all other AGAS services are available on `locality#0` only). | None | Returns the overall execution time of the specified AGAS service since its creation (in nanoseconds). |
| `/agas/<agas_service_category>/time`<br><br>where:<br>`<agas_service_category>` is one of the following: `primary`, `locality`, `component`, or `symbol`. | `<agas_instance>/total`<br><br>where:<br>`<agas_instance>` is the name of the AGAS service to query. Currently, this value will be `locality#0`, where `0` is the root locality (the id of the locality hosting the AGAS service). Except for `<agas_service_category>` `primary` or `symbol` for which the value for `*` can be any locality id (only the primary and symbol AGAS service components live on all localities, whereas all other AGAS services are available on `locality#0` only). | None | Returns the overall execution time of all AGAS services provided by the given AGAS service category since its creation (in nanoseconds). |
| `/agas/count/<cache_statistics>` | `locality#*/total`<br><br>where: | None | Returns the number of cache events (evictions, hits, inserts, and misses) in the AGAS cache |

| **Counter Type** | **Counter Instance Formatting** | **Parameters** | **Description** |
|---|---|---|---|
| where:<br>`<cache_statistics>` is one of the following: `cache-evictions`, `cache-hits`, `cache-inserts`, `cache-misses` | `*` is the locality id of the locality the AGAS cache should be queried. The locality id is a (zero based) number identifying the locality. | | of the specified locality (see `<cache_statistics>`. |

**Table 20. Parcel Layer Performance Counters**

| **Counter Type** | **Counter Instance Formatting** | **Parameters** | **Description** |
|---|---|---|---|
| `/data/count/<connection_type>/<operation>`<br><br>where:<br>`<operation>` is one of the following: `sent`, `received`<br>`<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the overall number of transmitted bytes should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall number of raw (uncompressed) bytes sent or received (see `<operation>`, e.g. `sent` or `received`) for the specified `<connection_type>`.<br><br>The performance counters for the connection type `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>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).<br><br>Please see Options and Variables for more details. |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/data/time/<connection_type>/<operation>`<br><br>where:<br>`<operation>` is one of the following: `sent`, `received`<br>`<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the total transmission time should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the total time (in nanoseconds) between the start of each asynchronous transmission operation and the end of the corresponding operation for the specified `<connection_type>` the given locality (see `<operation>`, e.g. `sent` or `received`).<br><br>The performance counters for the connection type `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>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).<br><br>Please see Options and Variables for more details. |
| `/serialize/count/<connection_type>/<operation>`<br><br>where:<br>`<operation>` is one of the following: `sent`, `received`<br>`<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the overall number of transmitted bytes should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall number of bytes transferred (see `<operation>`, e.g. `sent` or `received`, possibly compressed) for the specified `<connection_type>` by the given locality.<br><br>The performance counters for the connection type |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | | | `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>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).<br><br>Please see Options and Variables for more details. |
| `/serialize/time/<connection_type>/<operation>`<br><br>where:<br>`<operation>` is one of the following: `sent`, `received`<br>`<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the serialization time should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall time spent performing outgoing data serialization for the specified `<connection_type>`on the given locality (see `<operation>`, e.g. `sent` or `received`).<br><br>The performance counters for the connection type `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` was defined while compiling |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | | | the *HPX* core library (which is not defined by default).<br><br>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).<br><br>Please see Options and Variables for more details. |
| `/security/time/<connection_type>/<operation>`<br><br>where:<br>`<operation>` is one of the following: `sent`, `received`<br>`<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the time spent for security related operation should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall time spent performing outgoing security operations for the specified `<connection_type>`on the given locality (see `<operation>`, e.g. `sent` or `received`).<br><br>The performance counters for the connection type `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>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).<br><br>These performance counters are available only if |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | | | the compile time constant `HPX_HAVE_SECURITY` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>Please see Options and Variables for more details. |
| `/parcels/count/routed` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of routed parcels should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall number of routed (outbound) parcels transferred by the given locality.<br><br>Routed parcels are those which cannot directly be delivered to its destination as the local AGAS is not able to resolve the destination address. In this case a parcel is sent to the AGAS service component which is responsible for creating the destination GID (and is responsible for resolving the destination address). This AGAS service component will deliver the parcel to its final target. |
| `/parcels/count/<connection_type>/<operation>`<br><br>where:<br>`<operation>` is one of the following: `sent`, `received` `<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of parcels should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall number of parcels transferred using the specified `<connection_type>` by the given locality (see `<operation>`, e.g. `sent` or `received`).<br><br>The performance counters for the connection type `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` |

| **Counter Type** | **Counter Instance Formatting** | **Parameters** | **Description** |
|---|---|---|---|
| | | | was defined while compiling the *HPX* core library (which is not defined by default).<br><br>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).<br><br>Please see Options and Variables for more details. |
| `/messages/count/<connection_type>/<operation>`<br><br>where:<br>`<operation>` is one of the following: `sent`, `received`<br>`<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of messages should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall number of messages [a] transferred using the specified `<connection_type>` by the given locality (see `<operation>`, e.g. `sent` or `received`).<br><br>The performance counters for the connection type `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>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). |

| **Counter Type** | **Counter Instance Formatting** | **Parameters** | **Description** |
|---|---|---|---|
| | | | Please see Options and Variables for more details. |
| `/parcelport/count/<connection_type>/<cache_statistics>`<br><br>where:<br>`<cache_statistics>` is one of the following: `cache-insertions`, `cache-evictions`, `cache-hits`, `cache-misses`, `cache-misses`<br>`<connection_type>` is one of the following: `tcp`, `ipc`, `ibverbs`, `mpi` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of messages should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the overall number cache events (evictions, hits, inserts, misses, and reclaims) for the connection cache of the given connection type on the given locality (see `<cache_statistics>`, e.g. `cache-insertions`, `cache-evictions`, `cache-hits`, `cache-misses`, or `cache-reclaims`).<br><br>The performance counters for the connection type `ipc` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IPC` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>The performance counters for the connection type `ibverbs` are available only if the compile time constant `HPX_HAVE_PARCELPORT_IBVERBS` was defined while compiling the *HPX* core library (which is not defined by default).<br><br>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).<br><br>Please see Options and Variables for more details. |
| `/parcelqueue/length/<operation>`<br><br>where:<br>`<operation>` is one of the following: `send`, `receive` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the parcel queue should be queried. The locality | None | Returns the current number of parcels stored in the parcel queue (see `<operation>` for which queue to query, e.g. `send` or `receive`). |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | id is a (zero based) number identifying the locality. | | |

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

## Table 21. Thread Manager Performance Counters

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/threads/count/cumulative` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the overall number of retired *HPX*-threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality.<br><br>`worker-thread#*` is defining the worker thread for which the overall number of retired *HPX*-threads should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | None | Returns the overall number of executed (retired) *HPX*-threads on the given locality since application start. If the instance name is `total` the counter returns the accumulated number of retired *HPX*-threads for all worker threads (cores) on that locality. If the instance name is `worker-thread#*` the counter will return the overall number of retired *HPX*-threads for all worker threads separately. This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_CUMULATIVE_` is set to `ON` (default: ON). |
| `/threads/count/cumulative-phases` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the overall number of executed *HPX*-thread phases (invocations) should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality. | None | Returns the overall number of executed *HPX*-thread phases (invocations) on the given locality since application start. If the instance name is `total` the counter returns the accumulated number of executed *HPX*-thread phases (invocations) for all worker threads (cores) on that locality. If the instance name is `worker-thread#*` the counter |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | `worker-thread#*` is defining the worker thread for which the overall number of executed *HPX*-thread phases (invocations) should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | | will return the overall number of executed *HPX*-thread phases for all worker threads separately. This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_CUMULATIVE_` is set to `ON` (default: ON). |
| `/threads/count/instantaneous/<thread-state>`<br><br>where:<br>`<thread-state>` is one of the following: `all`, `active`, `pending`, `suspended`, `terminated`, `staged` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the overall number of retired *HPX*-threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`worker-thread#*` is defining the worker thread for which the overall number of retired *HPX*-threads should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`.<br><br>The `staged` thread state refers to registered tasks before they are converted to thread objects. | None | Returns the current number of *HPX*-threads having the given thread state on the given locality. If the instance name is `total` the counter returns the current number of *HPX*-threads of the given state for all worker threads (cores) on that locality. If the instance name is `worker-thread#*` the counter will return the current number of *HPX*-threads in the given state for all worker threads separately. |
| `/threads/wait-time/<thread-state>`<br><br>where: | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where: | None | 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 |

| **Counter Type** | **Counter Instance Formatting** | **Parameters** | **Description** |
|---|---|---|---|
| `<thread-state>` is one of the following: `pending`, `staged` | `locality#*` is defining the locality for which the overall number of retired *HPX*-threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`worker-thread#*` is defining the worker thread for which the overall number of retired *HPX*-threads should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`.<br><br>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. | | given locality since application start. If the instance name is `total` the counter returns the wait time of *HPX*-threads of the given state for all worker threads (cores) on that locality. If the instance name is `worker-thread#*` the counter will return the wait time of *HPX*-threads in the given state for all worker threads separately.<br><br>These counters are available only if the compile time constant `HPX_THREAD_MAINTAIN_QUEUE_WAITT` was defined while compiling the *HPX* core library (default: OFF). |
| `/threads/idle-rate` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`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 | None | Returns the average idle rate for the given worker thread(s) on the given locality. The idle rate is defined as the ratio of the time spent on scheduling and management tasks and the overall time spent executing work since the application started. This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_IDLE_RATES` is set to `ON` (default: OFF)/. |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | | |
| `/threads/creation-idle-rate` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`worker-thread#*` is defining the worker thread for which the averaged idle rate should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | None | Returns the average idle rate for the given worker thread(s) on the given locality which is caused by creating new threads. The creation idle rate is defined as the ratio of the time spent on creating new threads and the overall time spent executing work since the application started. This counter is available only if the configuration time constants `HPX_THREAD_MAINTAIN_IDLE_RATES` (default: OFF) and `HPX_THREAD_MAINTAIN_CREATION_AN` are set to `ON`. |
| `/threads/cleanup-idle-rate` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`worker-thread#*` is defining the worker thread for which the averaged idle rate should be queried for. The worker thread number | None | Returns the average idle rate for the given worker thread(s) on the given locality which is caused by cleaning up terminated threads. The cleanup idle rate is defined as the ratio of the time spent on cleaning up terminated thread objects and the overall time spent executing work since the application started. This counter is available only if the configuration time constants `HPX_THREAD_MAINTAIN_IDLE_RATES` (default: OFF) and `HPX_THREAD_MAINTAIN_CREATION_AN` are set to `ON`. |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | (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`. | | |
| `/threadqueue/length` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the current length of all thread queues in the scheduler for all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality.<br><br>`worker-thread#*` is defining the worker thread for which the current length of all thread queues in the scheduler should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | None | Returns the overall length of all queues for the given worker thread(s) on the given locality. |
| `/threads/count/stack-unbinds` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the unbind (madvise) operations should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the total number of *HPX*-thread unbind (madvise) operations performed for the referenced locality. Note that this counter is not available on Windows based platforms. |

| **Counter Type** | **Counter Instance Formatting** | **Parameters** | **Description** |
|---|---|---|---|
| `/threads/count/stack-recycles` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the unbind (madvise) operations should be queried for. The locality id is a (zero based) number identifying the locality. | None | Returns the total number of *HPX*-thread recycling operations performed. |
| `/threads/count/stolen-from-pending` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of 'stole' threads should be queried for. The locality id is a (zero based) number identifying the locality. | None | 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_THREAD_MAINTAIN_STEALING_CO` is set to `ON` (default: ON). |
| `/threads/count/pending-misses` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`worker-thread#*` is defining the worker thread for which the averaged idle rate should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | None | Returns the total number of times that the referenced worker-thread on the referenced locality failed to find pending HPX-threads in its associated queue. This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_STEALING_CO` is set to `ON` (default: ON). |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/threads/count/pending-accesses` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`worker-thread#*` is defining the worker thread for which the averaged idle rate should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | None | Returns the total number of times that the referenced worker-thread on the referenced locality looked for pending HPX-threads in its associated queue. This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_STEALING_CO` is set to `ON` (default: ON). |
| `/threads/count/stolen-from-staged` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`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 | None | Returns the total number of *HPX*-threads 'stolen' from the staged thread queue by a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_STEALING_CO` is set to `ON` (default: ON). |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | application using the option `--hpx:threads`. | | |
| `/threads/count/stolen-to-pending` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`worker-thread#*` is defining the worker thread for which the averaged idle rate should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | None | Returns the total number of *HPX*-threads 'stolen' from the pending thread queue of a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_STEALING_CO` is set to `ON` (default: ON). |
| `/threads/count/stolen-to-staged` | `locality#*/total` or `locality#*/worker-thread#*`<br><br>where:<br>`locality#*` is defining the locality for which the average idle rate of all (or one) worker threads should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality<br><br>`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 | None | Returns the total number of *HPX*-threads 'stolen' from the staged thread queue of a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant `HPX_THREAD_MAINTAIN_STEALING_CO` is set to `ON` (default: ON). |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. | | |
| `/threads/count/objects` | `locality#*/total` or `locality#*/allocator#*`<br><br>where:<br>`locality#*` is defining the locality for which the current (cumulative) number of all created *HPX*-thread objects should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality.<br><br>`allocator#*` is defining the number of the allocator instance using which the threads have been created. *HPX* uses a varying number of allocators to create (and recycle) *HPX*-thread objects, most likely these counters are of use for debugging purposes only. The allocator id (given by `*`) is a (zero based) number identifying the allocator to query. | None | Returns the total number of *HPX*-thread objects created. Note that thread objects are reused to improve system performance, thus this number does not reflect the number of actually executed (retired) *HPX*-threads. |

## Table 22. Performance Counters related to LCOs

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/full_empty/count/<operation>`<br><br>where:<br>`<operation>` is one of the following: `constructed`, `destructed`, `read_enqueued`, `read_dequeued`, `fired` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of invocations of the specified operation should be queried for (see `<operation>`, e.g. `constructed`, `destructed`, `read_enqueued`, `read_dequeued`, `fired`). The locality id is a (zero | None | Returns the overall number of operations performed on full-empty bit data structures on the specified locality. |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| | based) number identifying the locality. | | |
| `/dataflow/count/<operation>`<br><br>where:<br>`<operation>` is one of the following: `constructed`, `destructed`, `initialized`, `fired` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of invocations of the specified operation should be queried for (see `<operation>`, e.g. `constructed`, `destructed`, `initialized`, `fired`). The locality id is a (zero based) number identifying the locality. | None | Returns the overall number of operations performed on dataflow components on the specified locality. |

## Table 23. General Performance Counters exposing Characteristics of Localities

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/runtime/count/component` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the number of components should be queried. The locality id is a (zero based) number identifying the locality. | The type of the component. This is the string which has been used while registering the component with *HPX*, e.g. which has been passed as the second parameter to the macro `HPX_REGISTER_MINIMAL_COMPONENT_FACTORY`. | Returns the overall number of currently active components of the specified type on the given locality. |
| `/runtime/uptime` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the system uptime should be queried. The locality id is a (zero based) number identifying the locality. | None | Returns the overall time since application start on the given locality in nanoseconds. |
| `/runtime/memory/virtual` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the allocated virtual memory should be queried. The locality id is a (zero based) number identifying the locality. | None | Returns the amount of virtual memory currently allocated by the referenced locality (in bytes). |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/runtime/memory/resident` | `locality#*/total`<br><br>where:<br>`*` is the locality id of the locality the the allocated resident memory should be queried. The locality id is a (zero based) number identifying the locality. | None | Returns the amount of resident memory currently allocated by the referenced locality (in bytes). |

## Table 24. Performance Counters for General Statistics

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/statistics/average` | Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter. | Returns the current average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name). | Any parameter will be interpreted as 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. |
| `/statistics/stddev` | Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter. | Returns the current standard deviation (stddev) value calculated based on the values queried from the underlying counter (the one specified as the instance name). | Any parameter will be interpreted as 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. |
| `/statistics/rolling_average` | Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter. | Returns the current rolling average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name). | Any parameter will be interpreted as a list of 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 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`. |

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/statistics/median` | Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter. | Returns the current (statistically estimated) median value calculated based on the values queried from the underlying counter (the one specified as the instance name). | Any parameter will be interpreted as 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. |
| `/statistics/max` | Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter. | Returns the current maximum value calculated based on the values queried from the underlying counter (the one specified as the instance name). | Any parameter will be interpreted as 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. |
| `/statistics/min` | Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter. | Returns the current minimum value calculated based on the values queried from the underlying counter (the one specified as the instance name). | Any parameter will be interpreted as 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. |

### Table 25. Performance Counters for Elementary Arithmetic Operations

| Counter Type | Counter Instance Formatting | Parameters | Description |
|---|---|---|---|
| `/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. |

| **Counter Type** | **Counter Instance Formatting** | **Parameters** | **Description** |
|---|---|---|---|
| `/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. |

**Note**

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

```
./bin/hello_world --t2 \
    ---hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
    ---hpx:print-counter=/arithmetics/add@/threads{locality#0/worker-thread#*}/count/
cumulative
hello world from OS-thread 0 on locality 0
hello world from OS-thread 1 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.515640,[s],25
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.515520,[s],36
/arithmetics/add@/threads{locality#0/worker-thread#*}/count/cumulative,1,0.516445,
[s],64
```

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

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

## *HPX* Thread Scheduling Policies

The HPX runtime has seven thread scheduling policies: priority_local, local, global, abp, abp-priority, hierarchy and periodic 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_LOCAL_SCHEDULER=ON`, see Options and Variables for more information).

## Priority Local Scheduling Policy (default policy)

- default or invoke using: `--hpx:queuing=priority_local` (or `-qpr`)

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.

## Static Priority Scheduling Policy

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

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_LOCAL_SCHEDULER`

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

## Global Scheduling Policy

- invoke using: `--hpx:queuing global` or -qg
- flag to turn on for build: `HPX_GLOBAL_SCHEDULER`

The global scheduling policy maintains one shared queue from which all OS threads pull tasks.

## ABP Scheduling Policy

- invoke using: `--hpx:queuing=abp` (or `-qa`)
- flag to turn on for build: `HPX_ABP_SCHEDULER`

The ABP scheduling policy maintains a double ended lock free queue for each OS thread. Threads are pushed on the top of the queue, and during work stealing threads are taken from the bottom of the queue.

## Priority ABP Scheduling Policy

- invoke using: `--hpx:queuing=priority_abp`
- flag to turn on for build: `HPX_ABP_PRIORITY_SCHEDULER`

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-senstive`. 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.

## Hierarchy Scheduling Policy

- invoke using: `--hpx:queuing=hierarchy` (or `-qh`)
- flag to turn on for build: `HPX_HIERARCHY_SCHEDULER`

The hierarchy policy maintains a tree of work items. Every OS thread walks the tree to obtain new work. Arity of the thread queue tree can be specified on the command line using `--hpx:hierarchy-arity` (default is 2). Work stealing is done from the parent queue in that tree.

## Periodic Priority Scheduling Policy

- invoke using: `--hpx:queuing=periodic` (or `-qpe`)
- flag to turn on for build: `HPX_PERIODIC_PRIORITY_SCHEDULER`

Maintains one queue of work items (user threads) for each OS thread. Maintains a number of high priority queues (specified by `--hpx:high-priority-threads`) and one low priority queue. High priority threads are executed by the specified number of OS threads before any other work is executed. Low priority threads are executed when no other work is available.

# Index

A B C D E F G H I K L M N O P R S T U V Y

A

Action Invocation
- apply
- async
- async_continue

apply
- Action Invocation
- Applying an Action Asynchronously with Synchronization
- Applying an Action Asynchronously without any Synchronization
- Applying an Action with a Continuation but without any Synchronization
- Comparing Actions with C++ Functions
- Comparing C++ Functions and HPX Actions

Applying an Action Asynchronously with Synchronization
- apply
- async

Applying an Action Asynchronously without any Synchronization
- apply

Applying an Action with a Continuation and with Synchronization
- async
- async_continue

Applying an Action with a Continuation but without any Synchronization
- apply

assertion_failure
- Type error

async
- Action Invocation
- Applying an Action Asynchronously with Synchronization
- Applying an Action with a Continuation and with Synchronization
- Comparing C++ Functions and HPX Actions

async_continue
- Action Invocation
- Applying an Action with a Continuation and with Synchronization

B

bad_action_code
- Type error

bad_component_type
- Type error

bad_function_call
- Type error

bad_parameter
- Type error

bad_plugin_type
- Type error

bad_request
- Type error

bad_response_type
- Type error

broadcast
- Header < hpx/lcos/broadcast.hpp >

broadcast_apply

Header < hpx/lcos/broadcast.hpp >
broadcast_apply_with_index
Header < hpx/lcos/broadcast.hpp >
broadcast_with_index
Header < hpx/lcos/broadcast.hpp >
broken_promise
Type error
broken_task
Type error

C Class error_code
error_code
Class exception
exception
commandline_option_error
Type error
Comparing Actions with C++ Functions
apply
Comparing C++ Functions and HPX Actions
apply
async
copy
Function template copy

D deadlock
Type error
duplicate_component_address
Type error
duplicate_component_id
Type error
duplicate_console
Type error
dynamic_link_failure
Type error

E error
Type error
error_code
Class error_code
evaluate_active_counters
Function evaluate_active_counters
exception
Class exception

F filesystem_error
Type error
find_all_ids_from_basename
Function find_all_ids_from_basename
find_here
Function find_here

find_ids_from_basename
Function find_ids_from_basename
find_locality

find_root_locality
  Function find_root_locality
fold
  Header < hpx/lcos/fold.hpp >
fold_with_index
  Header < hpx/lcos/fold.hpp >
Function evaluate_active_counters
  evaluate_active_counters
Function find_all_ids_from_basename
  find_all_ids_from_basename
Function find_here
  find_here
Function find_ids_from_basename
  find_ids_from_basename
Function find_locality
  find_locality
Function find_root_locality
  find_root_locality
Function get_colocation_id
  get_colocation_id
Function get_colocation_id_sync
  get_colocation_id
Function get_locality_id
  get_locality_id
Function get_num_worker_threads
  get_num_worker_threads
Function get_system_uptime
  get_system_uptime
Function get_thread_name
  get_thread_name
Function get_worker_thread_num
  get_worker_thread_num
Function init
  init
Function is_running
  is_running
Function register_id_with_basename
  register_id_with_basename
Function register_shutdown_function
  register_shutdown_function
Function register_startup_function
  register_startup_function
Function reset_active_counters
  reset_active_counters
Function start
  start
Function start_active_counters
  start
  start_active_counters
Function stop
  stop
Function stop_active_counters
  stop
  stop_active_counters

Function template copy
copy
Function template get_ptr
get_ptr
Function template get_ptr_sync
get_ptr
Function template migrate
migrate
Function template new_
new_
Function template new_colocated
new_colocated
Function terminate
terminate
future_already_retrieved
Type error
future_can_not_be_cancelled
Type error
future_does_not_support_cancellation
Type error

G get_colocation_id
Function get_colocation_id
Function get_colocation_id_sync
get_locality_id
Function get_locality_id
get_num_worker_threads
Function get_num_worker_threads
get_ptr
Function template get_ptr
Function template get_ptr_sync
get_system_uptime
Function get_system_uptime
get_thread_name
Function get_thread_name
get_worker_thread_num
Function get_worker_thread_num

H Header < hpx/exception.hpp >
make_error_code
make_success_code
throwmode
throws
Header < hpx/lcos/broadcast.hpp >
broadcast
broadcast_apply
broadcast_apply_with_index
broadcast_with_index
Header < hpx/lcos/fold.hpp >
fold
fold_with_index
inverse_fold

reduce
reduce_with_index
Header < hpx/runtime/naming/unmanaged.hpp >
unmanaged
HPX_DEFINE_COMPONENT_ACTION
Macro HPX_DEFINE_COMPONENT_ACTION
HPX_DEFINE_COMPONENT_CONST_ACTION
Macro HPX_DEFINE_COMPONENT_CONST_ACTION
HPX_DEFINE_PLAIN_ACTION
Macro HPX_DEFINE_PLAIN_ACTION
HPX_PLAIN_ACTION
Macro HPX_PLAIN_ACTION
HPX_REGISTER_ACTION
Macro HPX_REGISTER_ACTION
HPX_REGISTER_ACTION_DECLARATION
Macro HPX_REGISTER_ACTION_DECLARATION
HPX_REGISTER_ACTION_DECLARATION_TEMPLATE
Macro HPX_REGISTER_ACTION_DECLARATION_TEMPLATE
HPX_REGISTER_PLAIN_ACTION
Macro HPX_REGISTER_PLAIN_ACTION
HPX_THROWS_IF
Macro HPX_THROWS_IF
HPX_THROW_EXCEPTION
Macro HPX_THROW_EXCEPTION

I
init
Function init
internal_server_error
Type error
invalid_data
Type error
invalid_status
Type error
inverse_fold
Header < hpx/lcos/fold.hpp >
inverse_fold_with_index
Header < hpx/lcos/fold.hpp >
is_running
Function is_running

K
kernel_error
Type error

L
lock_error
Type error

M
Macro HPX_DEFINE_COMPONENT_ACTION
HPX_DEFINE_COMPONENT_ACTION
Macro HPX_DEFINE_COMPONENT_CONST_ACTION
HPX_DEFINE_COMPONENT_CONST_ACTION
Macro HPX_DEFINE_PLAIN_ACTION
HPX_DEFINE_PLAIN_ACTION
Macro HPX_PLAIN_ACTION
HPX_PLAIN_ACTION
Macro HPX_REGISTER_ACTION

HPX_REGISTER_ACTION
Macro HPX_REGISTER_ACTION_DECLARATION
HPX_REGISTER_ACTION_DECLARATION
Macro HPX_REGISTER_ACTION_DECLARATION_TEMPLATE
HPX_REGISTER_ACTION_DECLARATION_TEMPLATE
Macro HPX_REGISTER_PLAIN_ACTION
HPX_REGISTER_PLAIN_ACTION
Macro HPX_THROWS_IF
HPX_THROWS_IF
Macro HPX_THROW_EXCEPTION
HPX_THROW_EXCEPTION
make_error_code
Header < hpx/exception.hpp >
make_success_code
Header < hpx/exception.hpp >
migrate
Function template migrate

N network_error
Type error
new_
Function template new_
new_colocated
Function template new_colocated
not_implemented
Type error
no_registered_console
Type error
no_state
Type error
no_success
Type error
null_thread_id
Type error

O out_of_memory
Type error

P promise_already_satisfied
Type error

R reduce
Header < hpx/lcos/reduce.hpp >
reduce_with_index
Header < hpx/lcos/reduce.hpp >
register_id_with_basename
Function register_id_with_basename
register_shutdown_function
Function register_shutdown_function
register_startup_function
Function register_startup_function
repeated_request
Type error
reset_active_counters
Function reset_active_counters

S
security_error
Type error
serialization_error
Type error
service_unavailable
Type error
start
Function start
Function start_active_counters
startup_timed_out
Type error
start_active_counters
Function start_active_counters
stop
Function stop
Function stop_active_counters
stop_active_counters
Function stop_active_counters
Struct thread_interrupted
thread_interrupted
success
Type error

T
task_already_started
Type error
task_moved
Type error
terminate
Function terminate
thread_interrupted
Struct thread_interrupted
thread_not_interruptable
Type error
thread_resource_error
Type error
throwmode
Header < hpx/exception.hpp >
throws
Header < hpx/exception.hpp >
Type error
assertion_failure
bad_action_code
bad_component_type
bad_function_call
bad_parameter
bad_plugin_type
bad_request
bad_response_type
broken_promise
broken_task
commandline_option_error
deadlock

duplicate_console
dynamic_link_failure
error
filesystem_error
future_already_retrieved
future_can_not_be_cancelled
future_does_not_support_cancellation
internal_server_error
invalid_data
invalid_status
kernel_error
lock_error
network_error
not_implemented
no_registered_console
no_state
no_success
null_thread_id
out_of_memory
promise_already_satisfied
repeated_request
security_error
serialization_error
service_unavailable
startup_timed_out
success
task_already_started
task_moved
thread_not_interruptable
thread_resource_error
unhandled_exception
uninitialized_value
unknown_component_address
unknown_error
version_too_new
version_too_old
version_unknown
yield_aborted

U unhandled_exception
Type error
uninitialized_value
Type error
unknown_component_address
Type error
unknown_error
Type error
unmanaged
Header < hpx/runtime/naming/unmanaged.hpp >

V version_too_new
Type error

version_unknown
Type error

Y
yield_aborted
Type error

# Reference

## Header <hpx/error.hpp>

```
namespace hpx {
  enum error;
}
```

### Type error

hpx::error — Possible error conditions.

## Synopsis

```
// In header: <hpx/error.hpp>


enum error { success = = 0, no_success = = 1, not_implemented = = 2, -
             out_of_memory = = 3, bad_action_code = = 4, -
             bad_component_type = = 5, network_error = = 6, -
             version_too_new = = 7, version_too_old = = 8, -
             version_unknown = = 9, unknown_component_address = = 10, -
             duplicate_component_address = = 11, invalid_status = = 12, -
             bad_parameter = = 13, internal_server_error = = 14, -
             service_unavailable = = 15, bad_request = = 16, -
             repeated_request = = 17, lock_error = = 18, -
             duplicate_console = = 19, no_registered_console = = 20, -
             startup_timed_out = = 21, uninitialized_value = = 22, -
             bad_response_type = = 23, deadlock = = 24, -
             assertion_failure = = 25, null_thread_id = = 26, -
             invalid_data = = 27, yield_aborted = = 28, -
             dynamic_link_failure = = 29, commandline_option_error = = 30, -
             serialization_error = = 31, unhandled_exception = = 32, -
             kernel_error = = 33, broken_task = = 34, task_moved = = 35, -
             task_already_started = = 36, future_already_retrieved = = 37, -
             promise_already_satisfied = = 38, -
             future_does_not_support_cancellation = = 39, -
             future_can_not_be_cancelled = = 40, no_state = = 41, -
             broken_promise = = 42, thread_resource_error = = 43, -
             future_cancelled = = 44, thread_cancelled = = 45, -
             thread_not_interruptable = = 46, duplicate_component_id = = 47, -
             unknown_error = = 48, bad_plugin_type = = 49, -
             security_error = = 50, filesystem_error = = 51, -
             bad_function_call = = 52 };
```

### Description

This enumeration lists all possible error conditions which can be reported from any of the API functions.

| | |
|---|---|
| `success` | The operation was successful. |
| `no_success` | The operation did failed, but not in an unexpected manner. |
| `not_implemented` | The operation is not implemented. |

| | |
|---|---|
| `out_of_memory` | The operation caused a out of memory condition. |
| `bad_component_type` | The specified component type is not known or otherwise invalid. |
| `network_error` | A generic network error occurred. |
| `version_too_new` | The version of the network representation for this object is too new. |
| `version_too_old` | The version of the network representation for this object is too old. |
| `version_unknown` | The version of the network representation for this object is unknown. |
| `duplicate_component_address` | The given global id has already been registered. |
| `invalid_status` | The operation was executed in an invalid status. |
| `bad_parameter` | One of the supplied parameters is invalid. |
| `duplicate_console` | There is more than one console locality. |
| `no_registered_console` | There is no registered console locality available. |
| `null_thread_id` | Attempt to invoke a API function from a non-HPX thread. |
| `yield_aborted` | The yield operation was aborted. |
| `commandline_option_error` | One of the options given on the command line is erroneous. |
| `serialization_error` | There was an error during serialization of this object. |
| `unhandled_exception` | An unhandled exception has been caught. |
| `kernel_error` | The OS kernel reported an error. |
| `broken_task` | The task associated with this future object is not available anymore. |
| `task_moved` | The task associated with this future object has been moved. |
| `task_already_started` | The task associated with this future object has already been started. |
| `future_already_retrieved` | The future object has already been retrieved. |
| `promise_already_satisfied` | The value for this future object has already been set. |
| `future_does_not_support_cancellation` | The future object does not support cancellation. |
| `future_can_not_be_cancelled` | The future can't be canceled at this time. |
| `no_state` | The future object has no valid shared state. |
| `broken_promise` | The promise has been deleted. |
| `duplicate_component_id` | The component type has already been registered. |
| `unknown_error` | An unknown error occurred. |
| `bad_plugin_type` | The specified plugin type is not known or otherwise invalid. |
| `security_error` | An error occurred in the security component. |
| `filesystem_error` | The specified file does not exist or other filesystem related error. |
| `bad_function_call` | equivalent of std::bad_function_call |

## Header <hpx/exception.hpp>

```


HPX_THROW_EXCEPTION(errcode, f, msg)
HPX_THROWS_IF(ec, errcode, f, msg)
```

```
namespace hpx {
  class error_code;
  class exception;

  struct thread_interrupted;

  // Encode error category for new error_code.
  enum throwmode { plain = = 0, rethrow = = 1, lightweight = = 0x80 };

  error_code throws;  // Predefined error_code object used as -"throw on error" tag.
  // Returns a new error_code constructed from the given parameters.
  error_code make_error_code(error e, throwmode mode = plain);
  error_code make_error_code(error e, char const * func, char const * file, -
                             long line, throwmode mode = plain);

  // Returns error_code(e, msg, mode).
  error_code make_error_code(error e, char const * msg, -
                             throwmode mode = plain);
  error_code make_error_code(error e, char const * msg, char const * func, -
                             char const * file, long line, -
                             throwmode mode = plain);

  // Returns error_code(e, msg, mode).
  error_code make_error_code(error e, std::string const & msg, -
                             throwmode mode = plain);
  error_code make_error_code(error e, std::string const & msg, -
                             char const * func, char const * file, long line, -
                             throwmode mode = plain);
  error_code make_error_code(boost::exception_ptr const & e);

  // Returns generic HPX error category used for new errors.
  boost::system::error_category const & get_hpx_category();

  // Returns generic HPX error category used for errors re-thrown after the exception has been -
de-serialized.
  boost::system::error_category const & get_hpx_rethrow_category();

  // Returns error_code(hpx::success, -"success", mode).
  error_code make_success_code(throwmode mode = plain);
  std::string diagnostic_information(hpx::exception const &);
  std::string diagnostic_information(hpx::error_code const &);
  std::string get_error_what(hpx::exception const &);
  std::string get_error_what(hpx::error_code const &);
  boost::uint32_t get_error_locality_id(hpx::exception const &);
  boost::uint32_t get_error_locality_id(hpx::error_code const &);
  error get_error(hpx::exception const &);
  error get_error(hpx::error_code const &);
  std::string get_error_host_name(hpx::exception const &);
  std::string get_error_host_name(hpx::error_code const &);
  boost::int64_t get_error_process_id(hpx::exception const &);
  boost::int64_t get_error_process_id(hpx::error_code const &);
  std::string get_error_env(hpx::exception const &);
  std::string get_error_env(hpx::error_code const &);
  std::string get_error_function_name(hpx::exception const &);
  std::string get_error_function_name(hpx::error_code const &);
  std::string get_error_backtrace(hpx::exception const &);
  std::string get_error_backtrace(hpx::error_code const &);
  std::string get_error_file_name(hpx::exception const &);
  std::string get_error_file_name(hpx::error_code const &);
  int get_error_line_number(hpx::exception const &);
  int get_error_line_number(hpx::error_code const &);
  std::size_t get_error_os_thread(hpx::exception const &);
  std::size_t get_error_os_thread(hpx::error_code const &);
  std::size_t get_error_thread_id(hpx::exception const &);
  std::size_t get_error_thread_id(hpx::error_code const &);
  std::string get_error_thread_description(hpx::exception const &);
  std::string get_error_thread_description(hpx::error_code const &);
  std::string get_error_config(hpx::exception const &);
  std::string get_error_config(hpx::error_code const &);
```

```
}
```

## Class error_code

hpx::error_code — A hpx::error_code represents an arbitrary error condition.

## Synopsis

```
// In header: <hpx/exception.hpp>


class error_code : public error_code {
public:
  // construct/copy/destruct
  explicit error_code(throwmode = plain);
  explicit error_code(error, throwmode = plain);
  error_code(error, char const *, char const *, long, throwmode = plain);
  error_code(error, char const *, throwmode = plain);
  error_code(error, char const *, char const *, char const *, long, -
             throwmode = plain);
  error_code(error, std::string const &, throwmode = plain);
  error_code(error, std::string const &, char const *, char const *, long, -
             throwmode = plain);
  error_code(int, hpx::exception const &);
  explicit error_code(boost::exception_ptr const &);
  error_code& operator=(error_code const &);

  // public member functions
  std::string get_message() const;
  void clear();
};
```

### Description

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

#### `error_code` public construct/copy/destruct

1.
```
explicit error_code(throwmode mode = plain);
```

Construct an object of type `error_code`.

| | | |
|---|---|---|
| Parameters: | `mode` | The parameter `mode` specifies whether the constructed `hpx::error_code` belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*). |
| Throws: | nothing | |

2. 
```
explicit error_code(error e, throwmode mode = plain);
```

Construct an object of type `error_code`.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `mode` | The parameter `mode` specifies whether the constructed `hpx::error_code` belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*). |
| Throws: | nothing | |

3. 
```
error_code(error e, char const * func, char const * file, long line,
           throwmode mode = plain);
```

Construct an object of type `error_code`.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `file` | The file name of the code where the error was raised. |
| | `func` | The name of the function 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 | |

4. 
```
error_code(error e, char const * msg, throwmode mode = plain);
```

Construct an object of type `error_code`.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `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*). |
| | `msg` | The parameter `msg` holds the error message the new exception should encapsulate. |
| Throws: | std::bad_alloc (if allocation of a copy of the passed string fails). | |

5. 
```
error_code(error e, char const * msg, char const * func, char const * file,
           long line, throwmode mode = plain);
```

Construct an object of type `error_code`.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `file` | The file name of the code where the error was raised. |
| | `func` | The name of the function 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*). |
| | `msg` | The parameter `msg` holds the error message the new exception should encapsulate. |
| Throws: | std::bad_alloc (if allocation of a copy of the passed string fails). | |

6. 
```
error_code(error e, std::string const & msg, throwmode mode = plain);
```

Construct an object of type `error_code`.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `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*). |
| | `msg` | The parameter `msg` holds the error message the new exception should encapsulate. |
| Throws: | std::bad_alloc (if allocation of a copy of the passed string fails). | |

7.
```
error_code(error e, std::string const & msg, char const * func,
           char const * file, long line, throwmode mode = plain);
```

Construct an object of type `error_code`.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `file` | The file name of the code where the error was raised. |
| | `func` | The name of the function 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*). |
| | `msg` | The parameter `msg` holds the error message the new exception should encapsulate. |
| Throws: | std::bad_alloc (if allocation of a copy of the passed string fails). | |

8.
```
error_code(int err, hpx::exception const & e);
```

9.
```
explicit error_code(boost::exception_ptr const & e);
```

10.
```
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.

### `error_code` public member functions

1.
```
std::string get_message() const;
```

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

Throws: nothing

2.
```
void clear();
```

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

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

## Class exception

hpx::exception — A hpx::exception is the main exception type used by HPX to report errors.

## Synopsis

```
// In header: <hpx/exception.hpp>


class exception : public system_error {
public:
  // construct/copy/destruct
  explicit exception(error);
  explicit exception(boost::system::system_error const &);
  exception(error, char const *, throwmode = plain);
  exception(error, std::string const &, throwmode = plain);
  ~exception();

  // public member functions
  error get_error() const;
  error_code get_error_code(throwmode = plain) const;
};
```

### Description

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.

#### `exception` public construct/copy/destruct

1.
```
explicit exception(error e);
```

Construct a `hpx::exception` from a *hpx::error*.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |

2.
```
explicit exception(boost::system::system_error const & e);
```

Construct a `hpx::exception` from a boost::system_error.

3.
```
exception(error e, char const * msg, throwmode mode = plain);
```

Construct a `hpx::exception` from a *hpx::error* and an error message.

| | | |
|---|---|---|
| Parameters: | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `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*). |

| | |
|---|---|
| `msg` | The parameter `msg` holds the error message the new exception should encapsulate. |

4.
```
exception(error e, std::string const & msg, throwmode mode = plain);
```

Construct a `hpx::exception` from a *hpx::error* and an error message.

| Parameters: | | |
|---|---|---|
| | `e` | The parameter `e` holds the hpx::error code the new exception should encapsulate. |
| | `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*). |
| | `msg` | The parameter `msg` holds the error message the new exception should encapsulate. |

5.
```
~exception();
```

Destruct a `hpx::exception`

Throws: nothing

### `exception` public member functions

1.
```
error get_error() const;
```

The function *get_error()* returns the hpx::error code stored in the referenced instance of a `hpx::exception`. It returns the hpx::error code this exception instance was constructed from.

Throws: nothing

2.
```
error_code get_error_code(throwmode = plain) const;
```

The function *get_error_code()* returns a `hpx::error_code` which represents the same error condition as this `hpx::exception` instance.

## Struct thread_interrupted

hpx::thread_interrupted — A hpx::thread_interrupted is the exception type used by HPX to interrupt a running HPX thread.

## Synopsis

```
// In header: <hpx/exception.hpp>


struct thread_interrupted : public exception {
};
```

### Description

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()*.

## Global throws

hpx::throws — Predefined error_code object used as "throw on error" tag.

# Synopsis

```
// In header: <hpx/exception.hpp>

error_code throws;
```

### Description

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.

## Function diagnostic_information

hpx::diagnostic_information — Extract the diagnostic information embedded in the given exception and return a string holding a formatted message.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string diagnostic_information(hpx::exception const & e);
```

### Description

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()*

| | | |
|---|---|---|
| Parameters: | `e` | 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`* or *`hpx::error_code`*. |
| Returns: | The formatted string holding all of the available diagnostic information stored in the given exception instance. | |
| Throws: | std::bad_alloc (if any of the required allocation operations fail) | |

## Function diagnostic_information

hpx::diagnostic_information — Extract the diagnostic information embedded in the given exception and return a string holding a formatted message.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string diagnostic_information(hpx::error_code const & e);
```

### Description

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()*

| | | |
|---|---|---|
| Parameters: | `e` | 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` or `hpx::error_code`. |
| Returns: | The formatted string holding all of the available diagnostic information stored in the given exception instance. | |
| Throws: | std::bad_alloc (if any of the required allocation operations fail) | |

## Function get_error_what

hpx::get_error_what — Return the error message of the thrown exception.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_what(hpx::exception const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| Returns: | The error message stored in the exception If the exception instance does not hold this information, the function will return an empty string. | |

Throws: std::bad_alloc (if one of the required allocations fails)

## Function get_error_what

hpx::get_error_what — Return the locality id where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_what(hpx::error_code const & e);
```

### Description

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()*

| | |
|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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*. |
| Throws: | nothing |

## Function get_error_locality_id

hpx::get_error_locality_id — Return the locality id where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


boost::uint32_t get_error_locality_id(hpx::exception const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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*, *boost::exception*, or *boost::exception_ptr*. |
| 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*. | |
| Throws: | nothing | |

## Function get_error_locality_id

hpx::get_error_locality_id — Return the locality id where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


boost::uint32_t get_error_locality_id(hpx::error_code const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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*, *boost::exception*, or *boost::exception_ptr*. |
| 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*. | |
| Throws: | nothing | |

## Function get_error

hpx::get_error — Return the locality id where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


error get_error(hpx::exception const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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 *boost::exception_ptr*. |
| 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*. | |
| Throws: | nothing | |

## Function get_error

hpx::get_error — Return the locality id where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


error get_error(hpx::error_code const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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 *boost::exception_ptr*. |
| 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*. | |
| Throws: | nothing | |

## Function get_error_host_name

hpx::get_error_host_name — Return the hostname of the locality where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_host_name(hpx::exception const & e);
```

### Description

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:**

*hpx::diagnostic_information() hpx::get_error_process_id(), hpx::get_error_function_name(), hpx::get_error_file_name(), hpx::get_error_line_number(), hpx::get_error_os_thread(), hpx::get_error_thread_id(), hpx::get_error_thread_description(), hpx::get_error() hpx::get_error_backtrace(), hpx::get_error_env(), hpx::get_error_what(), hpx::get_error_config()*

| | |
|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| Returns: | The hostname of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return and empty string. |
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Function get_error_host_name

hpx::get_error_host_name — Return the hostname of the locality where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_host_name(hpx::error_code const & e);
```

### Description

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:**

*hpx::diagnostic_information() hpx::get_error_process_id(), hpx::get_error_function_name(), hpx::get_error_file_name(), hpx::get_error_line_number(), hpx::get_error_os_thread(), hpx::get_error_thread_id(), hpx::get_error_thread_description(), hpx::get_error() hpx::get_error_backtrace(), hpx::get_error_env(), hpx::get_error_what(), hpx::get_error_config()*

| | |
|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| Returns: | The hostname of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return and empty string. |

Throws: std::bad_alloc (if one of the required allocations fails)

## Function get_error_process_id

hpx::get_error_process_id — Return the (operating system) process id of the locality where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


boost::int64_t get_error_process_id(hpx::exception const & e);
```

### Description

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:**

*hpx::diagnostic_information(), hpx::get_error_host_name(), hpx::get_error_function_name(), hpx::get_error_file_name(), hpx::get_error_line_number(), hpx::get_error_os_thread(), hpx::get_error_thread_id(), hpx::get_error_thread_description(), hpx::get_error(), hpx::get_error_backtrace(), hpx::get_error_env(), hpx::get_error_what(), hpx::get_error_config()*

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`*, *boost::exception*, or *boost::exception_ptr*.

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.

Throws: nothing

## Function get_error_process_id

hpx::get_error_process_id — Return the (operating system) process id of the locality where the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


boost::int64_t get_error_process_id(hpx::error_code const & e);
```

### Description

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:**

*hpx::diagnostic_information(), hpx::get_error_host_name(), hpx::get_error_function_name(), hpx::get_error_file_name(), hpx::get_error_line_number(), hpx::get_error_os_thread(), hpx::get_error_thread_id(), hpx::get_error_thread_description(), hpx::get_error(), hpx::get_error_backtrace(), hpx::get_error_env(), hpx::get_error_what(), hpx::get_error_config()*

| | |
|---|---|
| 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*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | nothing |

## Function get_error_env

hpx::get_error_env — Return the environment of the OS-process at the point the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_env(hpx::exception const & e);
```

### Description

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()*

| | |
|---|---|
| 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*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Function get_error_env

hpx::get_error_env — Return the environment of the OS-process at the point the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_env(hpx::error_code const & e);
```

### Description

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()*

| | |
|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Function get_error_function_name

hpx::get_error_function_name — Return the function name from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_function_name(hpx::exception const & e);
```

### Description

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()*

| | |
|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Function get_error_function_name

hpx::get_error_function_name — Return the function name from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>
```

```
std::string get_error_function_name(hpx::error_code const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. | |
| Throws: | std::bad_alloc (if one of the required allocations fails) | |

## Function get_error_backtrace

hpx::get_error_backtrace — Return the stack backtrace from the point the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_backtrace(hpx::exception const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. | |
| Throws: | std::bad_alloc (if one of the required allocations fails) | |

## Function get_error_backtrace

hpx::get_error_backtrace — Return the stack backtrace from the point the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_backtrace(hpx::error_code const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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. | |
| Throws: | std::bad_alloc (if one of the required allocations fails) | |

## Function get_error_file_name

hpx::get_error_file_name — Return the (source code) file name of the function from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_file_name(hpx::exception const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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. | |

| | |
|---|---|
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Function get_error_file_name

hpx::get_error_file_name — Return the (source code) file name of the function from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_file_name(hpx::error_code const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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. | |
| Throws: | std::bad_alloc (if one of the required allocations fails) | |

## Function get_error_line_number

hpx::get_error_line_number — Return the line number in the (source code) file of the function from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


int get_error_line_number(hpx::exception const & e);
```

### Description

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()*

| | |
|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | nothing |

## Function get_error_line_number

hpx::get_error_line_number — Return the line number in the (source code) file of the function from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


int get_error_line_number(hpx::error_code const & e);
```

### Description

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()*

| | |
|---|---|
| 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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | nothing |

## Function get_error_os_thread

hpx::get_error_os_thread — Return the sequence number of the OS-thread used to execute HPX-threads from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::size_t get_error_os_thread(hpx::exception const & e);
```

### Description

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()*

| | |
|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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). |
| Throws: | nothing |

## Function get_error_os_thread

hpx::get_error_os_thread — Return the sequence number of the OS-thread used to execute HPX-threads from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::size_t get_error_os_thread(hpx::error_code const & e);
```

### Description

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()*

| | |
|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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). |
| Throws: | nothing |

## Function get_error_thread_id

hpx::get_error_thread_id — Return the unique thread id of the HPX-thread from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>

```

```
std::size_t get_error_thread_id(hpx::exception const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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_t(0). | |
| Throws: | nothing | |

## Function get_error_thread_id

hpx::get_error_thread_id — Return the unique thread id of the HPX-thread from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::size_t get_error_thread_id(hpx::error_code const & e);
```

### Description

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()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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_t(0). | |
| Throws: | nothing | |

## Function get_error_thread_description

hpx::get_error_thread_description — Return any additionally available thread description of the HPX-thread from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_thread_description(hpx::exception const & e);
```

### Description

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_what(), hpx::get_error_config()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |
| 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. | |
| Throws: | std::bad_alloc (if one of the required allocations fails) | |

## Function get_error_thread_description

hpx::get_error_thread_description — Return any additionally available thread description of the HPX-thread from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_thread_description(hpx::error_code const & e);
```

### Description

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_what(), hpx::get_error_config()*

| | | |
|---|---|---|
| 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`, *boost::exception*, or *boost::exception_ptr*. |

| | |
|---|---|
| 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. |
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Function get_error_config

hpx::get_error_config — Return the HPX configuration information point from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_config(hpx::exception const & e);
```

### Description

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:**

*hpx::diagnostic_information()*, *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error()*, *hpx::get_error_what()*, *hpx::get_error_thread_description()*

| | |
|---|---|
| Parameters: | `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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Function get_error_config

hpx::get_error_config — Return the HPX configuration information point from which the exception was thrown.

## Synopsis

```
// In header: <hpx/exception.hpp>


std::string get_error_config(hpx::error_code const & e);
```

### Description

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:**

*hpx::diagnostic_information()*, *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error()*, *hpx::get_error_what()*, *hpx::get_error_thread_description()*

| | |
|---|---|
| Parameters: | `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`*, *boost::exception*, or *boost::exception_ptr*. |
| 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. |
| Throws: | std::bad_alloc (if one of the required allocations fails) |

## Macro HPX_THROW_EXCEPTION

HPX_THROW_EXCEPTION — Throw a hpx::exception initialized from the given parameters.

## Synopsis

```
// In header: <hpx/exception.hpp>

HPX_THROW_EXCEPTION(errcode, f, msg)
```

### Description

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.

**Example:**

```
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");
}
```

## Macro HPX_THROWS_IF

HPX_THROWS_IF — Either throw a hpx::exception or initialize *hpx::error_code* from the given parameters.

## Synopsis

```
// In header: <hpx/exception.hpp>

HPX_THROWS_IF(ec, errcode, f, msg)
```

### Description

The macro *HPX_THROWS_IF* can be used to either throw a *hpx::exception* or to initialize a *hpx::error_code* from the given parameters. If &ec == &hpx::throws, the semantics of this macro are equivalent to *HPX_THROW_EXCEPTION*. If &ec != &hpx::throws, the *hpx::error_code* instance `ec` is initialized instead.

The parameter `errcode` holds the hpx::error code from which the new exception should be initialized. The parameter `f` is expected to hold the name of the function exception is thrown from and the parameter `msg` holds the error message the new exception should encapsulate.

## Header <hpx/hpx_finalize.hpp>

```
namespace hpx {
  int finalize(double, double = -1.0, error_code & = throws);
  int finalize(error_code & = throws);
  void terminate(error_code & = throws);
  int disconnect(double, double = -1.0, error_code & = throws);
  int disconnect(error_code & = throws);
  int stop(error_code & = throws);
}
```

### Function finalize

hpx::finalize — Main function to gracefully terminate the the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_finalize.hpp>


int finalize(double shutdown_timeout, double localwait = -1.0, -
             error_code & ec = throws);
```

### Description

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.

**Note**

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

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.

| | | |
|---|---|---|
| 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. |
| | `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. |
| | `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. |
| Returns: | This function will always return zero. | |

## Function finalize

hpx::finalize — Main function to gracefully terminate the the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_finalize.hpp>


int finalize(error_code & ec = throws);
```

### Description

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.

**Note**

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

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.

| | | |
|---|---|---|
| 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. | |

## Function terminate

hpx::terminate — Terminate any application non-gracefully.

## Synopsis

```
// In header: <hpx/hpx_finalize.hpp>


void terminate(error_code & ec = throws);
```

### Description

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**

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

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*.

| | | |
|---|---|---|
| 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. |

## Function disconnect

hpx::disconnect — Disconnect this locality from the application.

## Synopsis

```
// In header: <hpx/hpx_finalize.hpp>


int disconnect(double shutdown_timeout, double localwait = -1.0, -
               error_code & ec = throws);
```

### Description

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.

**Note**

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

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.

| | | |
|---|---|---|
| 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. |
| | `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. |
| | `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. |
| Returns: | This function will always return zero. | |

## Function disconnect

hpx::disconnect — Disconnect this locality from the application.

## Synopsis

```
// In header: <hpx/hpx_finalize.hpp>


int disconnect(error_code & ec = throws);
```

### Description

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.

**Note**

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

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.

| | | |
|---|---|---|
| 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. | |

## Function stop

hpx::stop — Stop the runtime system.

## Synopsis

```
// In header: <hpx/hpx_finalize.hpp>
```

```
int stop(error_code & ec = throws);
```

### Description

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`).

## Header <hpx/hpx_fwd.hpp>

```
namespace boost {
}namespace hpx {
  typedef std::function< void()> startup_function_type;
  typedef std::function< void()> shutdown_function_type;
  naming::id_type find_here(error_code & = throws);
  naming::id_type find_root_locality(error_code & = throws);
  std::vector< naming::id_type > find_all_localities(error_code & = throws);
  std::vector< naming::id_type > -
  find_all_localities(components::component_type, error_code & = throws);
  std::vector< naming::id_type > find_remote_localities(error_code & = throws);
  std::vector< naming::id_type > -
  find_remote_localities(components::component_type, error_code & = throws);
  naming::id_type -
  find_locality(components::component_type, error_code & = throws);
  std::vector< hpx::future< hpx::id_type > > -
  find_all_ids_from_basename(char const *, std::size_t);
  std::vector< hpx::future< hpx::id_type > > -
  find_ids_from_basename(char const *, std::vector< std::size_t > const &);
  hpx::future< hpx::id_type > -
  find_id_from_basename(char const *, std::size_t = ~0U);
  hpx::future< bool > -
  register_id_with_basename(char const *, hpx::id_type, std::size_t = ~0U);
  hpx::future< hpx::id_type > -
  unregister_id_with_basename(char const *, std::size_t = ~0U);
  boost::uint32_t get_num_localities_sync(error_code & = throws);
  boost::uint32_t get_initial_num_localities();
  lcos::future< boost::uint32_t > get_num_localities();
  boost::uint32_t -
  get_num_localities_sync(components::component_type, error_code & = throws);
  lcos::future< boost::uint32_t > -
  get_num_localities(components::component_type);
  void register_pre_startup_function(startup_function_type const &);
  void register_startup_function(startup_function_type const &);
  void register_pre_shutdown_function(shutdown_function_type const &);
  void register_shutdown_function(shutdown_function_type const &);
  std::size_t get_worker_thread_num();
  boost::uint32_t get_locality_id(error_code & = throws);
  bool is_running();
  std::string get_thread_name();
  std::size_t get_num_worker_threads();
  boost::uint64_t get_system_uptime();
  naming::id_type -
  get_colocation_id_sync(naming::id_type const &, error_code & = throws);
  lcos::future< naming::id_type > get_colocation_id(naming::id_type const &);
  std::string get_locality_name();
  future< std::string > get_locality_name(naming::id_type const &);
  void trigger_lco_event(naming::id_type const &);
  void trigger_lco_event(naming::id_type const &, naming::id_type const &);
  template<typename T> void set_lco_value(naming::id_type const &, T &&);
  template<typename T> -
    void set_lco_value(naming::id_type const &, T &&, naming::id_type const &);
  void set_lco_error(naming::id_type const &, boost::exception_ptr const &);
  void set_lco_error(naming::id_type const &, boost::exception_ptr &&);
  void set_lco_error(naming::id_type const &, boost::exception_ptr const &, -
                     naming::id_type const &);
  void set_lco_error(naming::id_type const &, boost::exception_ptr &&, -
                     naming::id_type const &);
  void start_active_counters(error_code & = throws);
  void reset_active_counters(error_code & = throws);
  void stop_active_counters(error_code & = throws);
  void evaluate_active_counters(bool = false, char const * = 0, -
                                error_code & = throws);
  parcelset::policies::message_handler * -
  create_message_handler(char const *, char const *, parcelset::parcelport *, -
                         std::size_t, std::size_t, error_code & = throws);
  util::binary_filter * -
  create_binary_filter(char const *, bool, util::binary_filter * = 0, -
                       error_code & = throws);
}
```

## Type definition startup_function_type

startup_function_type

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


typedef std::function< void()> startup_function_type;
```

### Description

The type of a function which is registered to be executed as a startup or pre-startup function.

## Type definition shutdown_function_type

shutdown_function_type

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


typedef std::function< void()> shutdown_function_type;
```

### Description

The type of a function which is registered to be executed as a shutdown or pre-shutdown function.

## Function find_here

hpx::find_here — Return the global id representing this locality.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


naming::id_type find_here(error_code & ec = throws);
```

### Description

The function *find_here()* can be used to retrieve the global id usable to refer to the current locality.

**Note**

Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

**Note**

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

This function will return meaningful results only if called from an HPX-thread. It will return *hpx::naming::invalid_id* otherwise.

**See Also:**

*hpx::find_all_localities(), hpx::find_locality()*

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| Returns: | The global id representing the locality this function has been called on. | |

## Function find_root_locality

hpx::find_root_locality — Return the global id representing the root locality.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


naming::id_type find_root_locality(error_code & ec = throws);
```

### Description

The function *find_root_locality()* can be used to retrieve the global id usable to refer to the root locality. The root locality is the locality where the main AGAS service is hosted.

**Note**

Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

**Note**

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

This function will return meaningful results only if called from an HPX-thread. It will return *hpx::naming::invalid_id* otherwise.

**See Also:**

*hpx::find_all_localities()*, *hpx::find_locality()*

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| Returns: | The global id representing the root locality for this application. | |

## Function find_all_localities

hpx::find_all_localities — Return the list of global ids representing all localities available to this application.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::vector< naming::id_type > find_all_localities(error_code & ec = throws);
```

### Description

The function *find_all_localities()* can be used to retrieve the global ids of all localities currently available to this application.

**Note**

Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

**Note**

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

This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

**See Also:**

*hpx::find_here(), hpx::find_locality()*

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| Returns: | The global ids representing the localities currently available to this application. | |

## Function find_all_localities

hpx::find_all_localities — Return the list of global ids representing all localities available to this application which support the given component type.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::vector< naming::id_type > -
find_all_localities(components::component_type type, error_code & ec = throws);
```

### Description

The function *find_all_localities()* can be used to retrieve the global ids of all localities currently available to this application which support the creation of instances of the given component type.

**Note**

Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

**Note**

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

This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

**See Also:**

*hpx::find_here(), hpx::find_locality()*

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `type` | [in] The type of the components for which the function should return the available localities. |
| 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. | |

## Function find_remote_localities

hpx::find_remote_localities — 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).

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::vector< naming::id_type > -
find_remote_localities(error_code & ec = throws);
```

### Description

The function *find_remote_localities()* can be used to retrieve the global ids of all remote localities currently available to this application (i.e. all localities except the current one).

**Note**

Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

**Note**

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

This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

**See Also:**

*hpx::find_here(), hpx::find_locality()*

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| Returns: | The global ids representing the remote localities currently available to this application. | |

## Function find_remote_localities

hpx::find_remote_localities — 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).

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::vector< naming::id_type > -
find_remote_localities(components::component_type type, -
                       error_code & ec = throws);
```

### Description

The function *find_remote_localities()* can be used to retrieve the global ids of all remote localities currently available to this application (i.e. all localities except the current one) which support the creation of instances of the given component type.

**Note**

Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

**Note**

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

This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

**See Also:**

*hpx::find_here(), hpx::find_locality()*

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `type` | [in] The type of the components for which the function should return the available remote localities. |
| Returns: | The global ids representing the remote localities currently available to this application. | |

## Function find_locality

hpx::find_locality — Return the global id representing an arbitrary locality which supports the given component type.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


naming::id_type -
find_locality(components::component_type type, error_code & ec = throws);
```

### Description

The function *find_locality()* can be used to retrieve the global id of an arbitrary locality currently available to this application which supports the creation of instances of the given component type.

**Note**

Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

**Note**

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

This function will return meaningful results only if called from an HPX-thread. It will return *hpx::naming::invalid_id* otherwise.

**See Also:**

*hpx::find_here(), hpx::find_all_localities()*

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `type` | [in] The type of the components for which the function should return any available locality. |
| 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::naming::invalid_id*. | |

## Function find_all_ids_from_basename

hpx::find_all_ids_from_basename — Return all registered ids from all localities from the given base name.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::vector< hpx::future< hpx::id_type > > 
find_all_ids_from_basename(char const * base_name, std::size_t num_ids);
```

### Description

This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

**Note**

The future will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

| | | |
|---|---|---|
| Parameters: | `base_name` | [in] The base name for which to retrieve the registered ids. |
| | `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. | |

## Function find_ids_from_basename

hpx::find_ids_from_basename — Return registered ids from the given base name and sequence numbers.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::vector< hpx::future< hpx::id_type > > 
find_ids_from_basename(char const * base_name, 
                       std::vector< std::size_t > const & ids);
```

### Description

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 future will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

| | | |
|---|---|---|
| Parameters: | `base_name` | [in] The base name for which to retrieve the registered ids. |
| | `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. | |

## Function find_id_from_basename

hpx::find_id_from_basename — Return registered id from the given base name and sequence number.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


hpx::future< hpx::id_type > -
find_id_from_basename(char const * base_name, std::size_t sequence_nr = ~0U);
```

### Description

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 will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

| | | |
|---|---|---|
| Parameters: | `base_name` | [in] The base name for which to retrieve the registered ids. |
| Returns: | A representing the id which was registered using the given base name and sequence numbers. | |

## Function register_id_with_basename

hpx::register_id_with_basename — Register the given id using the given base name.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


hpx::future< bool > -
register_id_with_basename(char const * base_name, hpx::id_type id, -
                              std::size_t sequence_nr = ~0U);
```

### Description

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. | |

## Function unregister_id_with_basename

hpx::unregister_id_with_basename — Unregister the given id using the given base name.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


hpx::future< hpx::id_type > -
unregister_id_with_basename(char const * base_name, -
                            std::size_t sequence_nr = ~0U);
```

### Description

The function unregisters the given ids using the provided base name.

| | | |
|---|---|---|
| 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_id_with_basename* before. |
| Returns: | A future representing the result of the un-registration operation itself. | |

## Function get_num_localities_sync

hpx::get_num_localities_sync — Return the number of localities which are currently registered for the running application.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


boost::uint32_t get_num_localities_sync(error_code & ec = throws);
```

### Description

The function *get_num_localities* returns the number of localities currently connected to the console.

**Note**

This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

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

**See Also:**

*hpx::find_all_localities_sync*, *hpx::get_num_localities*

## Function get_initial_num_localities

hpx::get_initial_num_localities — Return the number of localities which were registered at startup for the running application.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


boost::uint32_t get_initial_num_localities();
```

### Description

The function *get_initial_num_localities* returns the number of localities which were connected to the console at application startup.

**Note**

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

**See Also:**

*hpx::find_all_localities*, *hpx::get_num_localities*

## Function get_num_localities

hpx::get_num_localities — Asynchronously return the number of localities which are currently registered for the running application.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


lcos::future< boost::uint32_t > get_num_localities();
```

### Description

The function *get_num_localities* asynchronously returns the number of localities currently connected to the console. The returned future represents the actual result.

> **Note**
>
> This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

**See Also:**

*hpx::find_all_localities*, *hpx::get_num_localities*

## Function get_num_localities_sync

hpx::get_num_localities_sync — Return the number of localities which are currently registered for the running application.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


boost::uint32_t -
get_num_localities_sync(components::component_type t, -
                        error_code & ec = throws);
```

### Description

The function *get_num_localities* returns the number of localities currently connected to the console which support the creation of the given component type.

> **Note**
>
> This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.
>
> As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of `hpx::exception`.

**See Also:**

*hpx::find_all_localities*, *hpx::get_num_localities*

| Parameters: | | |
|---|---|---|
| | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `t` | The component type for which the number of connected localities should be retrieved. |

## Function get_num_localities

hpx::get_num_localities — Asynchronously return the number of localities which are currently registered for the running application.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


lcos::future< boost::uint32_t > -
get_num_localities(components::component_type t);
```

### Description

The function *get_num_localities* asynchronously returns the number of localities currently connected to the console which support the creation of the given component type. The returned future represents the actual result.

> **Note**
>
> This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

**See Also:**

*hpx::find_all_localities*, *hpx::get_num_localities*

Parameters: `t` The component type for which the number of connected localities should be retrieved.

## Function register_pre_startup_function

hpx::register_pre_startup_function — Add a function to be executed by a HPX thread before hpx_main but guaranteed before any startup function is executed (system-wide).

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void register_pre_startup_function(startup_function_type const & f);
```

### Description

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()*).

> **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.

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()*

Parameters: `f` [in] The function to be registered to run by an HPX thread as a pre-startup function.

## Function register_startup_function

hpx::register_startup_function — Add a function to be executed by a HPX thread before hpx_main but guaranteed after any pre-startup function is executed (system-wide).

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void register_startup_function(startup_function_type const & f);
```

### Description

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.

**Note**

If this function is called while the startup functions are being executed or after that point, it will raise a invalid_status exception.

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()*

Parameters: `f` [in] The function to be registered to run by an HPX thread as a startup function.

## Function register_pre_shutdown_function

hpx::register_pre_shutdown_function — Add a function to be executed by a HPX thread during *hpx::finalize()* but guaranteed before any shutdown function is executed (system-wide)

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void register_pre_shutdown_function(shutdown_function_type const & f);
```

### Description

Any of the functions registered with *register_pre_shutdown_function* are guaranteed to be executed by an HPX thread during the execution of *hpx::finalize()* before any of the registered shutdown functions are executed (see: *hpx::register_shutdown_function()*).

**Note**

If this function is called while the pre-shutdown functions are being executed, or after that point, it will raise a invalid_status exception.

**See Also:**

*hpx::register_shutdown_function()*

Parameters: `f` [in] The function to be registered to run by an HPX thread as a pre-shutdown function.

## Function register_shutdown_function

hpx::register_shutdown_function — Add a function to be executed by a HPX thread during *hpx::finalize()* but guaranteed after any pre-shutdown function is executed (system-wide)

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void register_shutdown_function(shutdown_function_type const & f);
```

### Description

Any of the functions registered with *register_shutdown_function* are guaranteed to be executed by an HPX thread during the execution of *hpx::finalize()* after any of the registered pre-shutdown functions are executed (see: *hpx::register_pre_shutdown_function()*).

**Note**

If this function is called while the shutdown functions are being executed, or after that point, it will raise a invalid_status exception.

**See Also:**

*hpx::register_pre_shutdown_function()*

Parameters: `f` [in] The function to be registered to run by an HPX thread as a shutdown function.

## Function get_worker_thread_num

hpx::get_worker_thread_num — Return the number of the current OS-thread running in the runtime instance the current HPX-thread is executed with.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>
```

```
std::size_t get_worker_thread_num();
```

### Description

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()*.
>
> This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

## Function get_locality_id

hpx::get_locality_id — Return the number of the locality this function is being called from.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


boost::uint32_t get_locality_id(error_code & ec = throws);
```

### Description

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

## Function is_running

hpx::is_running — Test whether the runtime system is currently running.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>
```

```
bool is_running();
```

### Description

This function returns whether the runtime system is currently running or not, e.g. whether the current state of the runtime system is *hpx::runtime::running*

**Note**

This function needs to be executed on a HPX-thread. It will return false otherwise.

## Function get_thread_name

hpx::get_thread_name — Return the name of the calling thread.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::string get_thread_name();
```

### Description

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>".

## Function get_num_worker_threads

hpx::get_num_worker_threads — Return the number of worker OS- threads used to execute HPX threads.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::size_t get_num_worker_threads();
```

### Description

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.

## Function get_system_uptime

hpx::get_system_uptime — Return the system uptime measure on the thread executing this call.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


boost::uint64_t get_system_uptime();
```

### Description

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.

## Function get_colocation_id_sync

hpx::get_colocation_id_sync — Return the id of the locality where the object referenced by the given id is currently located on.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


naming::id_type 
get_colocation_id_sync(naming::id_type const & id, error_code & ec = throws);
```

### Description

The function hpx::get_colocation_id() returns the id of the locality where the given object is currently located.

**Note**

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

**See Also:**

*hpx::get_colocation_id()*

| 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. |
| | `id` | [in] The id of the object to locate. |

## Function get_colocation_id

hpx::get_colocation_id — Asynchronously return the id of the locality where the object referenced by the given id is currently located on.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>
```

```
lcos::future< naming::id_type > get_colocation_id(naming::id_type const & id);
```

### Description

**See Also:**

*hpx::get_colocation_id_sync()*

## Function get_locality_name

hpx::get_locality_name — Return the name of the locality this function is called on.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


std::string get_locality_name();
```

### Description

This function returns the name for the locality on which this function is called.

**See Also:**

*future<std::string>* get_locality_name(naming::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.

## Function get_locality_name

hpx::get_locality_name — Return the name of the referenced locality.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


future< std::string > get_locality_name(naming::id_type const & id);
```

### Description

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. | |

## Function trigger_lco_event

hpx::trigger_lco_event — Trigger the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void trigger_lco_event(naming::id_type const & id);
```

### Description

| | | |
|---|---|---|
| Parameters: | `id` | [in] This represents the id of the LCO which should be triggered. |

## Function trigger_lco_event

hpx::trigger_lco_event — Trigger the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void trigger_lco_event(naming::id_type const & id, 
                       naming::id_type const & cont);
```

### Description

| | | |
|---|---|---|
| Parameters: | `cont` | [in] This represents the LCO to trigger after completion. |
| | `id` | [in] This represents the id of the LCO which should be triggered. |

## Function template set_lco_value

hpx::set_lco_value — Set the result value for the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


template<typename T> void set_lco_value(naming::id_type const & id, T && t);
```

### Description

| | | |
|---|---|---|
| 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. |

## Function template set_lco_value

hpx::set_lco_value — Set the result value for the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


template<typename T> 
  void set_lco_value(naming::id_type const & id, T && t, 
                     naming::id_type const & cont);
```

### Description

| | | |
|---|---|---|
| Parameters: | `cont` | [in] This represents the LCO to trigger after completion. |
| | `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. |

## Function set_lco_error

hpx::set_lco_error — Set the error state for the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void set_lco_error(naming::id_type const & id, boost::exception_ptr const & e);
```

### Description

| | | |
|---|---|---|
| Parameters: | `e` | [in] This is the error value which should be sent to the LCO. |
| | `id` | [in] This represents the id of the LCO which should receive the error value. |

## Function set_lco_error

hpx::set_lco_error — Set the error state for the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void set_lco_error(naming::id_type const & id, boost::exception_ptr && e);
```

### Description

| Parameters: | | |
|---|---|---|
| | `e` | [in] This is the error value which should be sent to the LCO. |
| | `id` | [in] This represents the id of the LCO which should receive the error value. |

### Function set_lco_error

hpx::set_lco_error — Set the error state for the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void set_lco_error(naming::id_type const & id, boost::exception_ptr const & e, -
                   naming::id_type const & cont);
```

#### Description

| Parameters: | | |
|---|---|---|
| | `cont` | [in] This represents the LCO to trigger after completion. |
| | `e` | [in] This is the error value which should be sent to the LCO. |
| | `id` | [in] This represents the id of the LCO which should receive the error value. |

### Function set_lco_error

hpx::set_lco_error — Set the error state for the LCO referenced by the given id.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void set_lco_error(naming::id_type const & id, boost::exception_ptr && e, -
                   naming::id_type const & cont);
```

#### Description

| Parameters: | | |
|---|---|---|
| | `cont` | [in] This represents the LCO to trigger after completion. |
| | `e` | [in] This is the error value which should be sent to the LCO. |
| | `id` | [in] This represents the id of the LCO which should receive the error value. |

### Function start_active_counters

hpx::start_active_counters — Start all active performance counters, optionally naming the section of code.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>
```

```
void start_active_counters(error_code & ec = throws);
```

### Description

**Note**

As long as *ec* is not pre-initialized to *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 active counters are those which have been specified on the command line while executing the application (see command line option --hpx:print-counter)

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |

## Function reset_active_counters

hpx::reset_active_counters — Resets all active performance counters.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void reset_active_counters(error_code & ec = throws);
```

### Description

**Note**

As long as *ec* is not pre-initialized to *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 active counters are those which have been specified on the command line while executing the application (see command line option --hpx:print-counter)

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |

## Function stop_active_counters

hpx::stop_active_counters — Stop all active performance counters.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>
```

```
void stop_active_counters(error_code & ec = throws);
```

### Description

**Note**

As long as *ec* is not pre-initialized to *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 active counters are those which have been specified on the command line while executing the application (see command line option --hpx:print-counter)

| Parameters: | | |
|---|---|---|
| | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |

## Function evaluate_active_counters

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

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


void evaluate_active_counters(bool reset = false, 
                              char const * description = 0, 
                              error_code & ec = throws);
```

### Description

**Note**

As long as *ec* is not pre-initialized to *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 output generated by this function is redirected to the destination specified by the corresponding command line options (see --hpx:print-counter-destination).

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

| Parameters: | | |
|---|---|---|
| | `description` | [in] this is an optional value naming the point in the code marked by the call to this function. |
| | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `reset` | [in] this is an optional flag allowing to reset the counter value after it has been evaluated. |

## Function create_message_handler

hpx::create_message_handler — Create an instance of a message handler plugin.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


parcelset::policies::message_handler * -
create_message_handler(char const * message_handler_type, char const * action, -
                       parcelset::parcelport * pp, std::size_t num_messages, -
                       std::size_t interval, error_code & ec = throws);
```

### Description

The function hpx::create_message_handler() creates an instance of a message handler plugin based on the parameters specified.

**Note**

As long as *ec* is not pre-initialized to *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: | `action` | |
| | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `interval` | |
| | `message_handler_type` | |
| | `num_messages` | |
| | `pp` | |

## Function create_binary_filter

hpx::create_binary_filter — Create an instance of a binary filter plugin.

## Synopsis

```
// In header: <hpx/hpx_fwd.hpp>


util::binary_filter * -
create_binary_filter(char const * binary_filter_type, bool compress, -
                     util::binary_filter * next_filter = 0, -
                     error_code & ec = throws);
```

### Description

**Note**

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

| | | |
|---|---|---|
| Parameters: | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |

## Header <hpx/hpx_init.hpp>

```
namespace hpx_startup {
}namespace hpx {
  int init(std::function< int(boost::program_options::variables_map &vm)> const &, -
           boost::program_options::options_description const &, int, char **, -
           std::vector< std::string > const &, -
           std::function< void()> const & = std::function< void()>(), -
           std::function< void()> const & = std::function< void()>(), -
           hpx::runtime_mode = hpx::runtime_mode_default);
  int init(int(*)(boost::program_options::variables_map &vm), -
           boost::program_options::options_description const &, int, char **, -
           std::function< void()> const & = std::function< void()>(), -
           std::function< void()> const & = std::function< void()>(), -
           hpx::runtime_mode = hpx::runtime_mode_default);
  int init(boost::program_options::options_description const &, int, char **, -
           std::function< void()> const & = std::function< void()>(), -
           std::function< void()> const & = std::function< void()>(), -
           hpx::runtime_mode = hpx::runtime_mode_default);
  int init(boost::program_options::options_description const &, int, char **, -
           std::vector< std::string > const &, -
           std::function< void()> const & = std::function< void()>(), -
           std::function< void()> const & = std::function< void()>(), -
           hpx::runtime_mode = hpx::runtime_mode_default);
  int init(int, char **, std::vector< std::string > const &, -
           hpx::runtime_mode = hpx::runtime_mode_default);
  int init(boost::program_options::options_description const &, int, char **, -
           hpx::runtime_mode);
  int init(std::string const & app_name, int argc = 0, char ** argv = 0, -
           hpx::runtime_mode mode = hpx::runtime_mode_default);
  int init(int = 0, char ** = 0, -
           hpx::runtime_mode = hpx::runtime_mode_default);
  int init(std::vector< std::string > const &, -
           hpx::runtime_mode = hpx::runtime_mode_default);
  int init(int(*)(boost::program_options::variables_map &vm) f, -
           std::string const & app_name, int argc, char ** argv, -
           hpx::runtime_mode mode = hpx::runtime_mode_default);
  int init(std::function< int(int, char **)> const & f, -
           std::string const & app_name, int argc, char ** argv, -
           hpx::runtime_mode mode = hpx::runtime_mode_default);
}
```

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>


int init(std::function< int(boost::program_options::variables_map &vm)> const & f, -
         boost::program_options::options_description const & desc_cmdline, -
         int argc, char ** argv, std::vector< std::string > const & cfg, -
         std::function< void()> const & startup = std::function< void()>(), -
         std::function< void()> const & shutdown = std::function< void()>(), -
```

```
        hpx::runtime_mode mode = hpx::runtime_mode_default);
```

## Description

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 parameter`mode`.

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

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>


int init(int(*)(boost::program_options::variables_map &vm) f, 
         boost::program_options::options_description const & desc_cmdline, 
         int argc, char ** argv, 
         std::function< void()> const & startup = std::function< void()>(), 
         std::function< void()> const & shutdown = std::function< void()>(), 
         hpx::runtime_mode mode = hpx::runtime_mode_default);
```

### Description

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 parameter`mode`.

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `desc_cmdline` | [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by *hpx::init* (see description below). |
| | `f` | [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| | `shutdown` | [in] A function to be executed inside an HPX thread while hpx::finalize is executed. If this parameter is not given no function will be executed. |
| | `startup` | [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed. |
| Returns: | The function returns the value, which has been returned from the user supplied `f`. | |

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>


int init(boost::program_options::options_description const & desc_cmdline, -
         int argc, char ** argv, -
         std::function< void()> const & startup = std::function< void()>(), -
         std::function< void()> const & shutdown = std::function< void()>(), -
         hpx::runtime_mode mode = hpx::runtime_mode_default);
```

### Description

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

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

**Note**

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

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `desc_cmdline` | [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by *hpx::init* (see description below). |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| | `shutdown` | [in] A function to be executed inside an HPX thread while hpx::finalize is executed. If this parameter is not given no function will be executed. |
| | `startup` | [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed. |
| Returns: | The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode). | |

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>


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

### Description

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

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

**Note**

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

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `cfg` | A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1') |
| | `desc_cmdline` | [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by *hpx::init* (see description below). |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| | `shutdown` | [in] A function to be executed inside an HPX thread while hpx::finalize is executed. If this parameter is not given no function will be executed. |
| | `startup` | [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed. |
| Returns: | The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode). | |

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>


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

### Description

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

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

**Note**

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

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `cfg` | A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1') |

| | | |
|---|---|---|
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| Returns: | The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode). | |

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>


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

### Description

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

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

**Note**

If the parameter `mode` is runtime_mode_default, the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `desc_cmdline` | [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by *hpx::init* (see description below). |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| Returns: | The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode). | |

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>
```

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

### Description

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 not command line arguments are passed, console mode is assumed.
>
> If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section 'HPX Command Line Options'.

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| Returns: | The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode). | |

## Function init

hpx::init — Main entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_init.hpp>


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

### Description

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 not command line arguments are passed, console mode is assumed.
>
> If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section 'HPX Command Line Options'.

| | | |
|---|---|---|
| Parameters: | `cfg` | A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1') |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| Returns: | The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode). | |

## Header <hpx/hpx_start.hpp>

```
namespace hpx_startup {
}namespace hpx {
  bool start(std::function< int(boost::program_options::variables_map &vm)> const &, -
             boost::program_options::options_description const &, int, -
             char **, std::vector< std::string > const &, -
             std::function< void()> const & = std::function< void()>(), -
             std::function< void()> const & = std::function< void()>(), -
             hpx::runtime_mode = hpx::runtime_mode_default);
  bool start(int(*)(boost::program_options::variables_map &vm), -
             boost::program_options::options_description const &, int, -
             char **, -
             std::function< void()> const & = std::function< void()>(), -
             std::function< void()> const & = std::function< void()>(), -
             hpx::runtime_mode = hpx::runtime_mode_default);
  bool start(boost::program_options::options_description const &, int, -
             char **, -
             std::function< void()> const & = std::function< void()>(), -
             std::function< void()> const & = std::function< void()>(), -
             hpx::runtime_mode = hpx::runtime_mode_default);
  bool start(boost::program_options::options_description const &, int, -
             char **, std::vector< std::string > const &, -
             std::function< void()> const & = std::function< void()>(), -
             std::function< void()> const & = std::function< void()>(), -
             hpx::runtime_mode = hpx::runtime_mode_default);
  bool start(int, char **, std::vector< std::string > const &, -
             hpx::runtime_mode = hpx::runtime_mode_default);
  bool start(boost::program_options::options_description const &, int, -
             char **, hpx::runtime_mode);
  bool start(std::string const & app_name, int argc = 0, char ** argv = 0, -
             hpx::runtime_mode mode = hpx::runtime_mode_default);
  bool start(int = 0, char ** = 0, -
             hpx::runtime_mode = hpx::runtime_mode_default);
  bool start(std::vector< std::string > const &, -
             hpx::runtime_mode = hpx::runtime_mode_default);
  bool start(int(*)(boost::program_options::variables_map &vm) f, -
             std::string const & app_name, int argc, char ** argv, -
             hpx::runtime_mode mode = hpx::runtime_mode_default);
}
```

### Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>


bool start(std::function< int(boost::program_options::variables_map &vm)> const & f, -
           boost::program_options::options_description const & desc_cmdline, -
           int argc, char ** argv, std::vector< std::string > const & cfg, -
           std::function< void()> const & startup = std::function< void()>(), -
           std::function< void()> const & shutdown = std::function< void()>(), -
           hpx::runtime_mode mode = hpx::runtime_mode_default);
```

## Description

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 immediatly 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 parameter`mode`.

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `cfg` | A list of configuration settings which will be added to the system configuration before the runtime instance is run. Each of the entries in this list must have the format of a fully defined key/value pair from an ini-file (for instance 'hpx.component.enabled=1') |
| | `desc_cmdline` | [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by *hpx::init* (see description below). |
| | `f` | [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| | `shutdown` | [in] A function to be executed inside an HPX thread while hpx::finalize is executed. If this parameter is not given no function will be executed. |
| | `startup` | [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed. |
| Returns: | The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise. | |

## Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>


bool start(int(*)(boost::program_options::variables_map &vm) f, 
           boost::program_options::options_description const & desc_cmdline, 
           int argc, char ** argv, 
           std::function< void()> const & startup = std::function< void()>(), 
           std::function< void()> const & shutdown = std::function< void()>(), 
           hpx::runtime_mode mode = hpx::runtime_mode_default);
```

### Description

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 immediatly 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 parameter`mode`.

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `desc_cmdline` | [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by *hpx::init* (see description below). |
| | `f` | [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| | `shutdown` | [in] A function to be executed inside an HPX thread while hpx::finalize is executed. If this parameter is not given no function will be executed. |
| | `startup` | [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed. |
| Returns: | The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise. | |

## Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>
```

```
bool start(boost::program_options::options_description const & desc_cmdline, -
           int argc, char ** argv, -
           std::function< void()> const & startup = std::function< void()>(), -
           std::function< void()> const & shutdown = std::function< void()>(), -
           hpx::runtime_mode mode = hpx::runtime_mode_default);
```

### Description

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 immediatly after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

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

#### Note

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

| | | |
|---|---|---|
| Parameters: | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `desc_cmdline` | [in] This parameter may hold the description of additional command line arguments understood by the application. These options will be prepended to the default command line options understood by *hpx::init* (see description below). |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |
| | `shutdown` | [in] A function to be executed inside an HPX thread while hpx::finalize is executed. If this parameter is not given no function will be executed. |
| | `startup` | [in] A function to be executed inside a HPX thread before `f` is called. If this parameter is not given no function will be executed. |
| Returns: | The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise. | |

## Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>


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

### Description

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 immediatly after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

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

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

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

## Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>


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

### Description

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 immediatly after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

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

**Note**

If the parameter `mode` is runtime_mode_default, the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

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

Returns: The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

## Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>


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

### Description

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 immediatly after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

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

**Note**

If the parameter `mode` is runtime_mode_default, the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

| Parameters: | | |
|---|---|---|
| | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |

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

Returns: The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

## Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>


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

### Description

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 immediatly 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 not command line arguments are passed, console mode is assumed.

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

| Parameters: | | |
|---|---|---|
| | `argc` | [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`). |
| | `argv` | [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`). |
| | `mode` | [in] The mode the created runtime environment should be initialized in. There has to be exactly one locality in each HPX application which is executed in console mode (*hpx::runtime_mode_console*), all other localities have to be run in worker mode (*hpx::runtime_mode_worker*). Normally this is set up automatically, but sometimes it is necessary to explicitly specify the mode. |

Returns: The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

## Function start

hpx::start — Main non-blocking entry point for launching the HPX runtime system.

## Synopsis

```
// In header: <hpx/hpx_start.hpp>


bool start(std::vector< std::string > const & cfg, -
           hpx::runtime_mode mode = hpx::runtime_mode_default);
```

### Description

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 immediatly 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 not command line arguments are passed, console mode is assumed.

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

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

## Header <hpx/lcos/broadcast.hpp>

```
namespace hpx {
  namespace lcos {
    template<typename Action, typename ArgN, ... > -
      hpx::future< std::vector< decltype(Action(hpx::id_type, ArgN,...))> > -
      broadcast(std::vector< hpx::id_type > const &, ArgN, ...);
    template<typename Action, typename ArgN, ... > -
      void broadcast_apply(std::vector< hpx::id_type > const &, ArgN, ...);
    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 &, ArgN, ...);
    template<typename Action, typename ArgN, ... > -
      void broadcast_apply_with_index(std::vector< hpx::id_type > const &, -
                                      ArgN, ...);
  }
}
```

## Function template broadcast

hpx::lcos::broadcast — Perform a distributed broadcast operation.

## Synopsis

```
// In header: <hpx/lcos/broadcast.hpp>


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, ...);
```

### Description

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: | `argN` | [in] Any number of arbitrary arguments (passed by by const reference) which will be forwarded to the action invocation. |
| | `ids` | [in] A list of global identifiers identifying the target objects for which the given action will be invoked. |
| Returns: | This function returns a future representing the result of the overall reduction operation. | |

## Function template broadcast_apply

hpx::lcos::broadcast_apply — Perform an asynchronous (fire&forget) distributed broadcast operation.

## Synopsis

```
// In header: <hpx/lcos/broadcast.hpp>


template<typename Action, typename ArgN, ... > 
  void broadcast_apply(std::vector< hpx::id_type > const & ids, ArgN argN, 
                       ...);
```

### Description

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

## Function template broadcast_with_index

hpx::lcos::broadcast_with_index — Perform a distributed broadcast operation.

## Synopsis

```
// In header: <hpx/lcos/broadcast.hpp>


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, -
                       ...);
```

### Description

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: | `argN` | [in] Any number of arbitrary arguments (passed by by const reference) which will be forwarded to the action invocation. |
| | `ids` | [in] A list of global identifiers identifying the target objects for which the given action will be invoked. |
| Returns: | This function returns a future representing the result of the overall reduction operation. | |

## Function template broadcast_apply_with_index

hpx::lcos::broadcast_apply_with_index — Perform an asynchronous (fire&forget) distributed broadcast operation.

## Synopsis

```
// In header: <hpx/lcos/broadcast.hpp>


template<typename Action, typename ArgN, ... > -
  void broadcast_apply_with_index(std::vector< hpx::id_type > const & ids, -
                                  ArgN argN, ...);
```

### Description

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

## Header <hpx/lcos/fold.hpp>

```
namespace hpx {
  namespace lcos {
    template<typename Action, typename FoldOp, typename Init, typename ArgN, -
             ... > -
      hpx::future< decltype(Action(hpx::id_type, ArgN,...))> -
      fold(std::vector< hpx::id_type > const &, FoldOp &&, Init &&, ArgN, ...);
    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 &, FoldOp &&, Init &&, -
                      ArgN, ...);
    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 &, FoldOp &&, Init &&, -
                   ArgN, ...);
    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 &, FoldOp &&, -
                              Init &&, ArgN, ...);
  }
}
```

### Function template fold

hpx::lcos::fold — Perform a distributed fold operation.

## Synopsis

```
// In header: <hpx/lcos/fold.hpp>


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, ...);
```

### Description

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: | `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. |
| | `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. |
| | `ids` | [in] A list of global identifiers identifying the target objects for which the given action will be invoked. |
| | `init` | [in] The initial value to be used for the folding operation |
| Returns: | This function returns a future representing the result of the overall folding operation. | |

## Function template fold_with_index

hpx::lcos::fold_with_index — Perform a distributed folding operation.

## Synopsis

```
// In header: <hpx/lcos/fold.hpp>


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, ...);
```

### Description

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: | `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. |
| | `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. |
| | `ids` | [in] A list of global identifiers identifying the target objects for which the given action will be invoked. |
| | `init` | [in] The initial value to be used for the folding operation |
| Returns: | This function returns a future representing the result of the overall folding operation. | |

## Function template inverse_fold

hpx::lcos::inverse_fold — Perform a distributed inverse folding operation.

## Synopsis

```
// In header: <hpx/lcos/fold.hpp>


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, ...);
```

### Description

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: | `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. |
| | `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. |
| | `ids` | [in] A list of global identifiers identifying the target objects for which the given action will be invoked. |
| | `init` | [in] The initial value to be used for the folding operation |
| Returns: | This function returns a future representing the result of the overall folding operation. | |

## Function template inverse_fold_with_index

hpx::lcos::inverse_fold_with_index — Perform a distributed inverse folding operation.

## Synopsis

```
// In header: <hpx/lcos/fold.hpp>


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, ...);
```

### Description

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: | `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. |
| | `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. |
| | `ids` | [in] A list of global identifiers identifying the target objects for which the given action will be invoked. |
| | `init` | [in] The initial value to be used for the folding operation |
| Returns: | This function returns a future representing the result of the overall folding operation. | |

## Header <hpx/lcos/reduce.hpp>

```
namespace hpx {
  namespace lcos {
    template<typename Action, typename ReduceOp, typename ArgN, ... > -
      hpx::future< decltype(Action(hpx::id_type, ArgN,...))> -
      reduce(std::vector< hpx::id_type > const &, ReduceOp &&, ArgN, ...);
    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 &, ReduceOp &&, -
                        ArgN, ...);
  }
}
```

### Function template reduce

hpx::lcos::reduce — Perform a distributed reduction operation.

## Synopsis

```
// In header: <hpx/lcos/reduce.hpp>


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, ...);
```

### Description

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: | `argN` | [in] Any number of arbitrary arguments (passed by by const reference) which will be forwarded to the action invocation. |
| | `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. |
| Returns: | This function returns a future representing the result of the overall reduction operation. | |

## Function template reduce_with_index

hpx::lcos::reduce_with_index — Perform a distributed reduction operation.

## Synopsis

```
// In header: <hpx/lcos/reduce.hpp>


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, ...);
```

### Description

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: | `argN` | [in] Any number of arbitrary arguments (passed by by const reference) which will be forwarded to the action invocation. |
| | `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. |
| Returns: | This function returns a future representing the result of the overall reduction operation. | |

## Header <hpx/performance_counters/manage_counter_type.hpp>

```
namespace hpx {
  namespace performance_counters {
    counter_status 
    install_counter_type(std::string const &, 
                         std::function< boost::int64_t(bool)> const &, 
                         std::string const & = "", std::string const & = "", 
                         error_code & = throws);
    void install_counter_type(std::string const &, counter_type, 
                              error_code & = throws);
    counter_status 
    install_counter_type(std::string const &, counter_type, 
                         std::string const &, std::string const & = "", 
                         boost::uint32_t = HPX_PERFORMANCE_COUNTER_V1, 
                         error_code & = throws);
    counter_status 
    install_counter_type(std::string const &, counter_type, 
                         std::string const &, 
                         std::function< create_counter_func > const &, 
                         std::function< discover_counters_func > const &, -
                         boost::uint32_t = HPX_PERFORMANCE_COUNTER_V1, -
                         std::string const & = "", error_code & = throws);
  }
}
```

## Function install_counter_type

hpx::performance_counters::install_counter_type — Install a new generic performance counter type in a way, which will uninstall it automatically during shutdown.

## Synopsis

```
// In header: <hpx/performance_counters/manage_counter_type.hpp>


counter_status -
install_counter_type(std::string const & name, -
                     std::function< boost::int64_t(bool)> const & counter_value, -
                     std::string const & helptext = "", -
                     std::string const & uom = "", error_code & ec = throws);
```

### Description

The function *install_counter_type* will register a new generic counter type based on the provided function. The counter type will be automatically unregistered during system shutdown. Any consumer querying any instance of this this counter type will cause the provided function to be called and the returned value to be exposed as the counter value.

The counter type is registered such that there can be one counter instance per locality. The expected naming scheme for the counter instances is: `'/objectname{locality#<*>/total}/countername'` where '<*>' is a zero based integer identifying the locality the counter is created on.

**Note**

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

**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: | | |
|---|---|---|
| | `counter_value` | [in] The function to call whenever the counter value is requested by a consumer. |
| | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `helptext` | [in, optional] A longer descriptive text shown to the user to explain the nature of the counters created from this type. |
| | `name` | [in] The global virtual name of the counter type. This name is expected to have the format /objectname/countername. |
| | `uom` | [in] The unit of measure for the new performance counter type. |

| | |
|---|---|
| Returns: | If successful, this function returns *status_valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*). |

## Function install_counter_type

hpx::performance_counters::install_counter_type — Install a new performance counter type in a way, which will uninstall it automatically during shutdown.

## Synopsis

```
// In header: <hpx/performance_counters/manage_counter_type.hpp>


void install_counter_type(std::string const & name, counter_type type, 
                          error_code & ec = throws);
```

### Description

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.

As long as *ec* is not pre-initialized to *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. |
| | `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. |
| Returns: | If successful, this function returns *status_valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*). | |

## Function install_counter_type

hpx::performance_counters::install_counter_type — Install a new performance counter type in a way, which will uninstall it automatically during shutdown.

## Synopsis

```
// In header: <hpx/performance_counters/manage_counter_type.hpp>


counter_status 
install_counter_type(std::string const & name, counter_type type, 
                     std::string const & helptext, 
                     std::string const & uom = "", -
                     boost::uint32_t version = HPX_PERFORMANCE_COUNTER_V1, -
                     error_code & ec = throws);
```

## Description

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.

As long as *ec* is not pre-initialized to *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. |
| | `helptext` | [in] A longer descriptive text shown to the user to explain the nature of the counters created from this type. |
| | `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. |
| | `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. |
| Returns: | If successful, this function returns *status_valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*). | |

## Function install_counter_type

hpx::performance_counters::install_counter_type — Install a new generic performance counter type in a way, which will uninstall it automatically during shutdown.

## Synopsis

```
// In header: <hpx/performance_counters/manage_counter_type.hpp>


counter_status -
install_counter_type(std::string const & name, counter_type type, -
                     std::string const & helptext, -
                     std::function< create_counter_func > const & create_counter, -
                     std::function< discover_counters_func > const & discover_counters, -
                     boost::uint32_t version = HPX_PERFORMANCE_COUNTER_V1, -
                     std::string const & uom = "", error_code & ec = throws);
```

## Description

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: | `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. |
| | `ec` | [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead. |
| | `helptext` | [in] A longer descriptive text shown to the user to explain the nature of the counters created from this type. |
| | `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. |
| | `version` | [in] The version of the counter type. This is currently expected to be set to HPX_PERFORMANCE_COUNTER_V1. |
| Returns: | If successful, this function returns *status_valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*). | |

## Header <hpx/runtime/actions/action_support.hpp>

```


HPX_REGISTER_ACTION_DECLARATION(...)
HPX_REGISTER_ACTION_DECLARATION_TEMPLATE(TEMPLATE, TYPE)
HPX_REGISTER_ACTION(...)
```

## Macro HPX_REGISTER_ACTION_DECLARATION

HPX_REGISTER_ACTION_DECLARATION — Declare the necessary component action boilerplate code.

## Synopsis

```
// In header: <hpx/runtime/actions/action_support.hpp>

HPX_REGISTER_ACTION_DECLARATION(...)
```

### Description

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 ':'.

**Example:**

```
namespace app
{
    // Define a simple component exposing one action -'print_greating'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::simple_component_base<server>
    {
        void print_greating ()
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }

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

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

**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.

## Macro HPX_REGISTER_ACTION_DECLARATION_TEMPLATE

HPX_REGISTER_ACTION_DECLARATION_TEMPLATE — Declare the necessary component action boilerplate code for actions taking template type arguments.

## Synopsis

```
// In header: <hpx/runtime/actions/action_support.hpp>

HPX_REGISTER_ACTION_DECLARATION_TEMPLATE(TEMPLATE, TYPE)
```

### Description

The macro *HPX_REGISTER_ACTION_DECLARATION_TEMPLATE* can be used to declare all the boilerplate code which is required for proper functioning of component actions in the context of HPX, if those actions take template type arguments.

The parameter *template* specifies the list of template type declarations for the action type. This argument has to be wrapped into an additional pair of parenthesis.

The parameter *action* is the type of the action to declare the boilerplate for. This argument has to be wrapped into an additional pair of parenthesis.

**Example:**

```
namespace app
{
    // Define a simple component exposing one action -'print_greating'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::simple_component_base<server>
    {
        template <typename T>
        void print_greating (T t)
        {
            hpx::cout << "Hey -" << t << ", how are you?\n" << hpx::flush;
        }

        // Component actions need to be declared, this also defines the
        // type -'print_greating_action' representing the action.

        // Actions with template arguments (like print_greating<>()
        // above) require special type definitions. The simplest way
        // to define such an action type is by deriving from the HPX
        // facility make_action:
        template <typename T>
        struct print_greating_action
          : hpx::actions::make_action<
                void (server::*)(T), &server::template print_greating<T>,
                print_greating_action<T> >
        {};
    };
}

// Declare boilerplate code required for each of the component actions.
HPX_REGISTER_ACTION_DECLARATION_TEMPLATE((template T), (app::server::print_greating_action<T>));
```

**Note**

This macro has to be used once for each of the component actions defined as above. 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.

## Macro HPX_REGISTER_ACTION

HPX_REGISTER_ACTION — Define the necessary component action boilerplate code.

## Synopsis

```
// In header: <hpx/runtime/actions/action_support.hpp>

HPX_REGISTER_ACTION(...)
```

### Description

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* 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. There is no need to use this macro for actions which have template type arguments (see *HPX_REGISTER_ACTION_DECLARATION_TEMPLATE*)

## Header <hpx/runtime/actions/component_non_const_action.hpp>

```


HPX_DEFINE_COMPONENT_ACTION(...)
HPX_DEFINE_COMPONENT_CONST_ACTION(...)
```

### Macro HPX_DEFINE_COMPONENT_ACTION

HPX_DEFINE_COMPONENT_ACTION — Registers a non-const member function of a component as an action type with HPX.

## Synopsis

```
// In header: <hpx/runtime/actions/component_non_const_action.hpp>

HPX_DEFINE_COMPONENT_ACTION(...)
```

### Description

The macro *HPX_DEFINE_COMPONENT_CONST_ACTION* can be used to register a non-const member function of a component as an action type named *action_type*.

The parameter *component* is the type of the component exposing the non-const 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.

**Example:**

```
namespace app
{
    // Define a simple component exposing one action -'print_greating'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::simple_component_base<server>
    {
        void print_greating ()
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }

        // Component actions need to be declared, this also defines the
        // type -'print_greating_action' representing the action.
        HPX_DEFINE_COMPONENT_ACTION(server, print_greating, print_greating_action);
    };
}
```

**Note**

This macro should be used for non-const member functions only. Use the macro *HPX_DEFINE_COMPONENT_CONST_ACTION* for const member functions.

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.

**Note**

The macro *HPX_DEFINE_COMPONENT_ACTION* can be used with 2 or 3 arguments. The third argument is optional.

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 first argument with an appended suffix '_action' resolves to a valid, unqualified C++ type name.

## Macro HPX_DEFINE_COMPONENT_CONST_ACTION

HPX_DEFINE_COMPONENT_CONST_ACTION — Registers a const member function of a component as an action type with HPX.

## Synopsis

```
// In header: <hpx/runtime/actions/component_non_const_action.hpp>

HPX_DEFINE_COMPONENT_CONST_ACTION(...)
```

### Description

The macro *HPX_DEFINE_COMPONENT_CONST_ACTION* can be used to register a const member function of a component as an action type named *action_type*.

The parameter *component* is the type of the component exposing the const 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.

**Example:**

```
namespace app
{
    // Define a simple component exposing one action -'print_greating'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::simple_component_base<server>
    {
        void print_greating() const
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }
        // Component actions need to be declared, this also defines the
        // type -'print_greating_action' representing the action.
        HPX_DEFINE_COMPONENT_CONST_ACTION(server, print_greating, print_greating_action);
    };
}
```

**Note**

This macro should be used for const member functions only. Use the macro *HPX_DEFINE_COMPONENT_ACTION* for non-const member functions.

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.

**Note**

The macro *HPX_DEFINE_COMPONENT_CONST_ACTION* can be used with 2 or 3 arguments. The third argument is optional.

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 first argument with an appended suffix '_action' resolves to a valid, unqualified C++ type name.

## Header <hpx/runtime/actions/plain_action.hpp>

```


HPX_REGISTER_PLAIN_ACTION(...)
HPX_REGISTER_PLAIN_ACTION_TEMPLATE(template_, action_type)
HPX_DEFINE_PLAIN_ACTION(func, name)
HPX_PLAIN_ACTION(...)
```

### Macro HPX_REGISTER_PLAIN_ACTION

HPX_REGISTER_PLAIN_ACTION — Registers an existing free function as a plain action with HPX.

## Synopsis

```
// In header: <hpx/runtime/actions/plain_action.hpp>

HPX_REGISTER_PLAIN_ACTION(...)
```

### Description

The macro *HPX_REGISTER_PLAIN_ACTION* can be used to register an existing free function as a plain action. It additionally defines an action type named *action_type*.

The parameter `action_type` is the name of the action type to register with HPX.

**Example:**

```
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);
}

// 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 `app::some_global_function`.
//
// The second argument used has to be the same as used for the
// HPX_DEFINE_PLAIN_ACTION above.
HPX_REGISTER_PLAIN_ACTION(app::some_global_action, some_global_action);
```

**Note**

Usually this macro will not be used in user code unless the intend is to avoid defining the action_type in global namespace. Normally, the use of the macro *HPX_PLAIN_ACTION* is recommend.

## Macro HPX_REGISTER_PLAIN_ACTION_TEMPLATE

HPX_REGISTER_PLAIN_ACTION_TEMPLATE — Registers an existing template-free function as a plain action with HPX.

## Synopsis

```
// In header: <hpx/runtime/actions/plain_action.hpp>

HPX_REGISTER_PLAIN_ACTION_TEMPLATE(template_, action_type)
```

### Description

The macro *HPX_REGISTER_PLAIN_ACTION* can be used to register an existing template-free function as a plain action. It relies on a separately defined action type named *action_type*.

The parameter `action_type` is the name of the action type to register with HPX.

**Example:**

```
namespace app
{
    template <typename T>
    void some_global_function(T d)
    {
        cout << d;
    }

    // define a new template action type named some_global_action
    template <typename T>
    struct some_global_action
      : hpx::actions::make_action<
            void (*)(T), &some_global_function<T>, -
            some_global_action<T> >
    {};
}

// 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 `app::some_global_function`.
//
// Please note the extra parenthesis around both macro arguments.
HPX_REGISTER_PLAIN_ACTION_TEMPLATE((template <typename T>), (app::some_global_action<T>));
```

## Macro HPX_DEFINE_PLAIN_ACTION

HPX_DEFINE_PLAIN_ACTION — Defines a plain action type.

## Synopsis

```
// In header: <hpx/runtime/actions/plain_action.hpp>

HPX_DEFINE_PLAIN_ACTION(func, name)
```

### Description

**Example:**

```
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);
}

// 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 `app::some_global_function`.
//
// The second argument used has to be the same as used for the
// HPX_DEFINE_PLAIN_ACTION above.
HPX_REGISTER_PLAIN_ACTION(app::some_global_action, some_global_action);
```

**Note**

Usually this macro will not be used in user code unless the intend is to avoid defining the action_type in global namespace. Normally, the use of the macro *HPX_PLAIN_ACTION* is recommend.

## Macro HPX_PLAIN_ACTION

HPX_PLAIN_ACTION — Defines a plain action type based on the given function *func* and registers it with HPX.

## Synopsis

```
// In header: <hpx/runtime/actions/plain_action.hpp>

HPX_PLAIN_ACTION(...)
```

### Description

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.

**Example:**

```
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);
```

**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.

The macro *HPX_PLAIN_ACTION* 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*.

## Header <hpx/runtime/components/copy_component.hpp>

```
namespace hpx {
  namespace components {
    template<typename Component> -
      future< naming::id_type > -
      copy(naming::id_type const &, -
           naming::id_type const & = naming::invalid_id);
  }
}
```

### Function template copy

hpx::components::copy — Copy given component to the specified target locality.

## Synopsis

```
// In header: <hpx/runtime/components/copy_component.hpp>


template<typename Component> 
  future< naming::id_type > 
  copy(naming::id_type const & to_copy, 
       naming::id_type const & target_locality = naming::invalid_id);
```

### Description

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**

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: | `target_locality` | [in, optional] The locality where the copy should be created (default is same locality as source). |
| | `to_copy` | [in] The global id of the component to copy |
| Returns: | A future representing the global id of the newly (copied) component instance. | |

## Header <hpx/runtime/components/migrate_component.hpp>

```
namespace hpx {
  namespace components {
    template<typename Component> 
      future< naming::id_type > 
      migrate(naming::id_type const &, naming::id_type const &);
  }
}
```

### Function template migrate

hpx::components::migrate — Migrate the given component to the specified target locality.

## Synopsis

```
// In header: <hpx/runtime/components/migrate_component.hpp>
```

```
template<typename Component> 
  future< naming::id_type > 
  migrate(naming::id_type const & to_migrate, 
          naming::id_type const & target_locality);
```

### Description

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.

**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: | `target_locality` | [in] The locality where the component should be migrated to. |
| | `to_migrate` | [in] The global id of the component to migrate. |
| Returns: | A future representing the global id of the newly (copied) component instance. | |

## Header <hpx/runtime/components/new.hpp>

```
namespace hpx {
  namespace components {
    template<typename Component, typename ArgN, ... > 
      hpx::future< hpx::id_type > new_(hpx::id_type const &, Arg0, ...);
    template<typename Component, typename ArgN, ... > 
      hpx::future< hpx::id_type > 
      new_colocated(hpx::id_type const &, Arg0, ...);
  }
}
```

### Function template new_

hpx::components::new_ — Create a new instance of the given Component type on the specified locality.

## Synopsis

```
// In header: <hpx/runtime/components/new.hpp>


template<typename Component, typename ArgN, ... > 
  hpx::future< hpx::id_type > 
  new_(hpx::id_type const & locality, Arg0 argN, ...);
```

### Description

This function creates a new instance 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.

| | | |
|---|---|---|
| Parameters: | `argN` | [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. |
| | `locality` | [in] The global address of the locality where the new instance should be created on. |
| Returns: | The function returns an *hpx::future* object instance which can be used to retrieve the global address of the newly created component. | |

## Function template new_colocated

hpx::components::new_colocated — Create a new instance of the given Component type on the co-located with the specified object.

## Synopsis

```
// In header: <hpx/runtime/components/new.hpp>


template<typename Component, typename ArgN, ... > - 
  hpx::future< hpx::id_type > - 
  new_colocated(hpx::id_type const & id, Arg0 argN, ...);
```

### Description

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

| | | |
|---|---|---|
| Parameters: | `argN` | [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. |
| | `id` | [in] The global address of an object defining the locality where the new instance should be created on. |
| Returns: | The function returns an *hpx::future* object instance which can be used to retrieve the global address of the newly created component. | |

## Header <hpx/runtime/get_ptr.hpp>

```
namespace hpx {
  template<typename Component> - 
    hpx::future< boost::shared_ptr< Component > > - 
    get_ptr(naming::id_type const &);
  template<typename Component> - 
    boost::shared_ptr< Component > - 
    get_ptr_sync(naming::id_type const &, error_code & = throws);
}
```

## Function template get_ptr

hpx::get_ptr — Returns a future referring to a the pointer to the underlying memory of a component.

## Synopsis

```
// In header: <hpx/runtime/get_ptr.hpp>
```

```
template<typename Component> 
  hpx::future< boost::shared_ptr< Component > > 
  get_ptr(naming::id_type const & id);
```

### Description

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 the calling locality. Otherwise the function will raise an error.

| | | |
|---|---|---|
| Parameters: | `id` | [in] The global id 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*. | |

## Function template get_ptr_sync

hpx::get_ptr_sync — Returns the pointer to the underlying memory of a component.

## Synopsis

```
// In header: <hpx/runtime/get_ptr.hpp>


template<typename Component> 
  boost::shared_ptr< Component > 
  get_ptr_sync(naming::id_type const & id, error_code & ec = throws);
```

### Description

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 the requesting locality. Otherwise the function will raise and error.
>
> As long as *ec* is not pre-initialized to *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. |
| | `id` | [in] The global id of the component for which the pointer to the underlying memory should be retrieved. |
| Returns: | This function returns the pointer to the underlying memory for the component instance with the given *id*. | |

## Header <hpx/runtime/naming/unmanaged.hpp>

```
namespace hpx {
  namespace naming {
    id_type unmanaged(id_type const &);
  }
}
```

## Function unmanaged

hpx::naming::unmanaged

## Synopsis

```
// In header: <hpx/runtime/naming/unmanaged.hpp>


id_type unmanaged(id_type const & id);
```

### Description

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. | |

# Terminology

This section gives definitions for some of the terms used throughout the *HPX* documentation and source code.

## Locality

A locality in *HPX* describes a synchronous domain of execution, or the domain of bounded upper response time. This normally is just a single node in a cluster or a NUMA domain in a SMP machine.

## Active Global Address Space (AGAS)

*HPX* incorporates a global address space. Any executing thread can access any object within the domain of the parallel application with the caveat that it must have appropriate access privileges. The model does not assume that global addresses are cache coherent; all loads and stores will deal directly with the site of the target object. All global addresses within a Synchronous Domain are assumed to be cache coherent for those processor cores that incorporate transparent caches. The Active Global Address Space used by *HPX* differs from research PGAS 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. It also incorporates capabilities based access rights for protection and security.

## Parcel

The Parcel is a component in *HPX* that communicates data, invokes an action at a distance, and distributes flow-control through the migration of continuations. Parcels bridge the gap of asynchrony between synchronous domains while maintaining symmetry of semantics between local and global execution. Parcels enable message-driven computation and may be seen as a form of "active messages". Other important forms of message-driven computation predating active messages include dataflow tokens, the J-machine's 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 (LCO)

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

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

# People

The STE||AR Group (pronounced as stellar) stands for "**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch". We are an international group of faculty, researchers, and students working at different organizations. The goal of the STE||AR 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.

All of our work is centered around building technologies for scalable parallel applications. *HPX*, our general purpose C++ runtime system for parallel and distributed applications, is no exeption. 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 way we 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 extention, these application will not be able to exploit high-end computing systems which are likely to employ hundreds of millions of such cores by the end of this decade.

Critical bottlenecks to the effective use of new generation high performance computing (HPC) systems include:

- *Starvation*: due to lack of usable application parallelism and means of managing it,
- *Overhead*: reduction to permit strong scalability, improve efficiency, and enable dynamic resource management,
- *Latency*: from remote access across system or to local memories,
- *Contention*: due to multicore chip I/O pins, memory banks, and system interconnects.

The ParalleX model has been devised to address these challenges by enabling a new computing dynamic through the application of message-driven computation in a global address space context with lightweight synchronization. The work on hpx is centered around implementing the concepts as defined by the ParalleX model. *HPX* is currently targetted 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 26. Contributors

| Name | Institution | email |
|---|---|---|
| Hartmut Kaiser | Center for Computation and Technology (CCT), Louisiana State University (LSU) | hkaiser@cct.lsu.edu |
| Thomas Heller | Department of Computer Science 3 - Computer Architecture, Friedrich-Alexander University Erlangen-Nuremberg (FAU) | thom.heller@gmail.com |
| Agustin Berge | Center for Computation and Technology (CCT), Louisiana State University (LSU) | kaballo86@hotmail.com |

| Name | Institution | email |
|---|---|---|
| Bryce Adelstein-Lelbach | Center for Computation and Technology (CCT), Louisiana State University (LSU) | blelbach@cct.lsu.edu |
| Vinay C Amatya | Center for Computation and Technology (CCT), Louisiana State University (LSU) | vamatya@cct.lsu.edu |
| Shuangyang Yang | Center for Computation and Technology (CCT), Louisiana State University (LSU) | syang16@cct.lsu.edu |
| Jeroen Habraken | Technische Universiteit Eindhoven | jhabraken@cct.lsu.edu |
| Steven Brandt | Center for Computation and Technology (CCT), Louisiana State University (LSU) | sbrandt@cct.lsu.edu |
| Andrew Kemp | Center for Computation and Technology (CCT), Louisiana State University (LSU) | akemp1@tigers.lsu.edu |
| Adrian Serio | Center for Computation and Technology (CCT), Louisiana State University (LSU) | aserio@cct.lsu.edu |
| Maciej Brodowicz | Center for Research in Extreme Scale Technologies (CREST), Indiana University (IU) | mbrodowi@indiana.edu |
| Matthew Anderson | Center for Research in Extreme Scale Technologies (CREST), Indiana University (IU) | matt@phys.lsu.edu |
| Alex Nagelberg | Center for Computation and Technology (CCT), Louisiana State University (LSU) | nalex@cct.lsu.edu |
| Dylan Stark | Sandia National Labs (Albuquerque) | dylan.stark@gmail.com |

## Contributors to this Document

**Table 27. Documentation Authors**

| Name | Institution | email |
|---|---|---|
| Hartmut Kaiser | Center for Computation and Technology (CCT), Louisiana State University (LSU) | hkaiser@cct.lsu.edu |
| Thomas Heller | Department of Computer Science 3 - Computer Architecture, Friedrich-Alexander University Erlangen-Nuremberg (FAU) | thom.heller@gmail.com |

| Name | Institution | email |
|---|---|---|
| Bryce Adelstein-Lelbach | Center for Computation and Technology (CCT), Louisiana State University (LSU) | blelbach@cct.lsu.edu |
| Vinay C Amatya | Center for Computation and Technology (CCT), Louisiana State University (LSU) | vamatya@cct.lsu.edu |
| Steven Brandt | Center for Computation and Technology (CCT), Louisiana State University (LSU) | sbrandt@cct.lsu.edu |
| Maciej Brodowicz | Center for Research in Extreme Scale Technologies (CREST), Indiana University (IU) | mbrodowi@indiana.edu |
| Adrian Serio | Center for Computation and Technology (CCT), Louisiana State University (LSU) | aserio@cct.lsu.edu |

## Acknowledgements

Thanks also to the following people who contributed directly or indirectly to the project through discussions, pull requests, documentation patches, etc.

- John Biddiscombe (CSCS - Swiss National Supercomputing Centre), who helped with the BlueGene/Q port of *HPX* and made several other contributions.
- Erik Schnetter (Perimeter Institute for Theoretical Physics), who greatly helped to make *HPX* more robust by sumbitting 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 fixing some documentation inconsistencies.
- Patricia Grubel (New Mexico State University), who contributed the desciption of the different *HPX* thread scheduler policies and is working on the performance analysis of our thread scheduling subsystem.
- Lars Viklund, who contributed platform specific patches for FreeBSD and MSVC12.
- Agustin Berge, who contributed patches fixing some very nasty hidden template meta-programming issues.
- Pyry Jahkola, who contributed the Mac OS build system and build documentation on how to build HPX using Clang and libc++.
- Mario Mulansky, who created an *HPX* backend for his Boost.Odeint library, and who submitted several test cases allowing us to reproduce and fix problems in *HPX*.
- Rekha Raj, who contributed changes to the description of the Windows build instructions.
- Alex Nagelberg for his work on implementing a C wrapper API for HPX.

In addition to the people who worked directly with *HPX* development we would like to acknowledge the NSF, DoE, DARPA, Center for Computation and Technology (CCT), and Department of Computer Science 3 - Computer Architecture who fund and support our work. We would also like to thank the following organizations for granting us allocatons of thier compute resources: LSU HPC, LONI, XSEDE and the Gauss Center for Supercomputing.

*HPX* is currently funded by:

The National Science Foundation through awards 1117470 (APX) and 1240655 (STAR). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

The Department of Energy (DoE) through the award DE-SC0008714 (XPRESS). Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

The Bavarian Research Foundation (Bayerische Forschungsstfitung) through the grant AZ-987-11.