Boost.MPI

Douglas Gregor

Matthias Troyer

Table of Contents

Introduction

Boost.MPI is a library for message passing in high-performance parallel applications. A Boost.MPI program is one or more processes that can communicate either via sending and receiving individual messages (point-to-point communication) or by coordinating as a group (collective communication). Unlike communication in threaded environments or using a shared-memory library, Boost.MPI processes can be spread across many different machines, possibly with different operating systems and underlying architectures.

Boost.MPI is not a completely new parallel programming library. Rather, it is a C++-friendly interface to the standard Message Passing Interface (MPI), the most popular library interface for high-performance, distributed computing. MPI defines a library interface, available from C, Fortran, and C++, for which there are many MPI implementations. Although there exist C++ bindings for MPI, they offer little functionality over the C bindings. The Boost.MPI library provides an alternative C++ interface to MPI that better supports modern C++ development styles, including complete support for user-defined data types and C++ Standard Library types, arbitrary function objects for collective algorithms, and the use of modern C++ library techniques to maintain maximal efficiency.

At present, Boost.MPI supports the majority of functionality in MPI 1.1. The thin abstractions in Boost.MPI allow one to easily combine it with calls to the underlying C MPI library. Boost.MPI currently supports:

- Communicators: Boost.MPI supports the creation, destruction, cloning, and splitting of MPI communicators, along with manipulation of process groups.
- Point-to-point communication: Boost.MPI supports point-to-point communication of primitive and user-defined data types with send and receive operations, with blocking and non-blocking interfaces.
- Collective communication: Boost.MPI supports collective operations such as `reduce` and `gather` with both built-in and user-defined data types and function objects.
- MPI Datatypes: Boost.MPI can build MPI data types for user-defined types using the Boost.Serialization library.
- Separating structure from content: Boost.MPI can transfer the shape (or "skeleton") of complexc data structures (lists, maps, etc.) and then separately transfer their content. This facility optimizes for cases where the data within a large, static data structure needs to be transmitted many times.

Boost.MPI can be accessed either through its native C++ bindings, or through its alternative, Python interface.

Getting started

Getting started with Boost.MPI requires a working MPI implementation, a recent version of Boost, and some configuration information.

MPI Implementation

To get started with Boost.MPI, you will first need a working MPI implementation. There are many conforming MPI implementations available. Boost.MPI should work with any of the implementations, although it has only been tested extensively with:

- Open MPI 1.0.x
- LAM/MPI 7.x
- MPICH 1.2.x

You can test your implementation using the following simple program, which passes a message from one processor to another. Each processor prints a message to standard output.

```
#include <mpi.h>
#include <iostream>

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    int value = 17;
    int result = MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    if (result == MPI_SUCCESS)
      std::cout << "Rank 0 OK!" << std::endl;
  } else if (rank == 1) {
    int value;
    int result = MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
     MPI_STATUS_IGNORE);
    if (result == MPI_SUCCESS && value == 17)
      std::cout << "Rank 1 OK!" << std::endl;
  }
  MPI_Finalize();
  return 0;
}
```

You should compile and run this program on two processors. To do this, consult the documentation for your MPI implementation. With LAM/MPI, for instance, you compile with the `mpiCC` or `mpic++` compiler, boot the LAM/MPI daemon, and run your program via `mpirun`. For instance, if your program is called `mpi-test.cpp`, use the following commands:

```
mpiCC -o mpi-test mpi-test.cpp
lamboot
mpirun -np 2 ./mpi-test
lamhalt
```

When you run this program, you will see both `Rank 0 OK!` and `Rank 1 OK!` printed to the screen. However, they may be printed in any order and may even overlap each other. The following output is perfectly legitimate for this MPI program:

```
Rank Rank 1 OK!
0 OK!
```

If your output looks something like the above, your MPI implementation appears to be working with a C++ compiler and we're ready to move on.

Configure and Build

Boost.MPI uses version 2 of the Boost.Build system for configuring and building the library binary. You will need a very new version of Boost.Jam (3.1.12 or later). If you already have Boost.Jam, run `bjam -v` to determine what version you are using.

Information about building Boost.Jam is available here. However, most users need only run `build.sh` in the `tools/build/jam_src` subdirectory of Boost. Then, copy the resulting `bjam` executable some place convenient.

For many users using LAM/MPI, MPICH, or OpenMPI, configuration is almost automatic. If you don't already have a file `user-config.jam` in your home directory, copy `tools/build/v2/user-config.jam` there. For many users, MPI support can be enabled simply by adding the following line to your user-config.jam file, which is used to configure Boost.Build version 2.

```
using mpi ;
```

This should auto-detect MPI settings based on the MPI wrapper compiler in your path, e.g., `mpic++`. If the wrapper compiler is not in your path, see below.

To actually build the MPI library, go into the top-level Boost directory and execute the command:

```
bjam --with-mpi
```

If your MPI wrapper compiler has a different name from the default, you can pass the name of the wrapper compiler as the first argument to the mpi module:

```
using mpi : /opt/mpich2-1.0.4/bin/mpiCC ;
```

If your MPI implementation does not have a wrapper compiler, or the MPI auto-detection code does not work with your MPI's wrapper compiler, you can pass MPI-related options explicitly via the second parameter to the `mpi` module:

```
using mpi : : <find-shared-library>lammpio <find-shared-library>lammpi++
              <find-shared-library>mpi <find-shared-library>lam
              <find-shared-library>dl ;
```

To see the results of MPI auto-detection, pass `--debug-configuration` on the bjam command line.

The (optional) fourth argument configures Boost.MPI for running regression tests. These parameters specify the executable used to launch jobs (default: "mpirun") followed by any necessary arguments to this to run tests and tell the program to expect the number of processors to follow (default: "-np"). With the default parameters, for instance, the test harness will execute, e.g.,

```
mpirun -np 4 all_gather_test
```

Installing and Using Boost.MPI

Installation of Boost.MPI can be performed in the build step by specifying `install` on the command line and (optionally) providing an installation location, e.g.,

```
bjam --with-mpi install
```

This command will install libraries into a default system location. To change the path where libraries will be installed, add the option `--prefix=PATH`.

To build applications based on Boost.MPI, compile and link them as you normally would for MPI programs, but remember to link against the `boost_mpi` and `boost_serialization` libraries, e.g.,

```
mpic++ -I/path/to/boost/mpi my_application.cpp -Llibdir \
  -lboost_mpi-gcc-mt-1_35 -lboost_serialization-gcc-d-1_35.a
```

If you plan to use the Python bindings for Boost.MPI in conjunction with the C++ Boost.MPI, you will also need to link against the boost_mpi_python library, e.g., by adding `-lboost_mpi_python-gcc-mt-1_35` to your link command. This step will only be necessary if you intend to register C++ types or use the skeleton/content mechanism from within Python.

Testing Boost.MPI

If you would like to verify that Boost.MPI is working properly with your compiler, platform, and MPI implementation, a self-contained test suite is available. To use this test suite, you will need to first configure Boost.Build for your MPI environment and then run `bjam` in `libs/mpi/test` (possibly with some extra options). For LAM/MPI, you will need to run `lamboot` before running `bjam`.

For MPICH, you may need to create a machine file and pass `-sMPIRUN_FLAGS="-machinefile <filename>"` to Boost.Jam; see the section on configuration for more information. If testing succeeds, `bjam` will exit without errors.

Tutorial

A Boost.MPI program consists of many cooperating processes (possibly running on different computers) that communicate among themselves by passing messages. Boost.MPI is a library (as is the lower-level MPI), not a language, so the first step in a Boost.MPI is to create an `mpi::environment` object that initializes the MPI environment and enables communication among the processes. The `mpi::environment` object is initialized with the program arguments (which it may modify) in your main program. The creation of this object initializes MPI, and its destruction will finalize MPI. In the vast majority of Boost.MPI programs, an instance of `mpi::environment` will be declared in `main` at the very beginning of the program.

Communication with MPI always occurs over a **communicator**, which can be created be simply default-constructing an object of type `mpi::communicator`. This communicator can then be queried to determine how many processes are running (the "size" of the communicator) and to give a unique number to each process, from zero to the size of the communicator (i.e., the "rank" of the process):

```
#include <boost/mpi/environment.hpp>
#include <boost/mpi/communicator.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;
  std::cout << "I am process " << world.rank() << " of " << world.size()
            << "." << std::endl;
  return 0;
}
```

If you run this program with 7 processes, for instance, you will receive output such as:

```
I am process 5 of 7.
I am process 0 of 7.
I am process 1 of 7.
I am process 6 of 7.
I am process 2 of 7.
I am process 4 of 7.
I am process 3 of 7.
```

Of course, the processes can execute in a different order each time, so the ranks might not be strictly increasing. More interestingly, the text could come out completely garbled, because one process can start writing "I am a process" before another process has finished writing "of 7.".

Point-to-Point communication

As a message passing library, MPI's primary purpose is to routine messages from one process to another, i.e., point-to-point. MPI contains routines that can send messages, receive messages, and query whether messages are available. Each message has a source process, a target process, a tag, and a payload containing arbitrary data. The source and target processes are the ranks of the sender and receiver of the message, respectively. Tags are integers that allow the receiver to distinguish between different messages coming from the same sender.

The following program uses two MPI processes to write "Hello, world!" to the screen (`hello_world.cpp`):

```
#include <boost/mpi.hpp>
#include <iostream>
#include <string>
#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  if (world.rank() == 0) {
    world.send(1, 0, std::string("Hello"));
    std::string msg;
    world.recv(1, 1, msg);
    std::cout << msg << "!" << std::endl;
  } else {
    std::string msg;
    world.recv(0, 0, msg);
    std::cout << msg << ", ";
    std::cout.flush();
    world.send(0, 1, std::string("world"));
  }

  return 0;
}
```

The first processor (rank 0) passes the message "Hello" to the second processor (rank 1) using tag 0. The second processor prints the string it receives, along with a comma, then passes the message "world" back to processor 0 with a different tag. The first processor then writes this message with the "!" and exits. All sends are accomplished with the `communicator::send` method and all receives use a corresponding `communicator::recv` call.

Non-blocking communication

The default MPI communication operations--`send` and `recv`--may have to wait until the entire transmission is completed before they can return. Sometimes this **blocking** behavior has a negative impact on performance, because the sender could be performing useful computation while it is waiting for the transmission to occur. More important, however, are the cases where several communication operations must occur simultaneously, e.g., a process will both send and receive at the same time.

Let's revisit our "Hello, world!" program from the previous section. The core of this program transmits two messages:

```
if (world.rank() == 0) {
  world.send(1, 0, std::string("Hello"));
  std::string msg;
  world.recv(1, 1, msg);
  std::cout << msg << "!" << std::endl;
} else {
  std::string msg;
  world.recv(0, 0, msg);
  std::cout << msg << ", ";
  std::cout.flush();
  world.send(0, 1, std::string("world"));
}
```

The first process passes a message to the second process, then prepares to receive a message. The second process does the send and receive in the opposite order. However, this sequence of events is just that--a **sequence**--meaning that there is essentially no parallelism. We can use non-blocking communication to ensure that the two messages are transmitted simultaneously (`hello_world_nonblocking.cpp`):

```
#include <boost/mpi.hpp>
#include <iostream>
#include <string>
#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  if (world.rank() == 0) {
    mpi::request reqs[2];
    std::string msg, out_msg = "Hello";
    reqs[0] = world.isend(1, 0, out_msg);
    reqs[1] = world.irecv(1, 1, msg);
    mpi::wait_all(reqs, reqs + 2);
    std::cout << msg << "!" << std::endl;
  } else {
    mpi::request reqs[2];
    std::string msg, out_msg = "world";
    reqs[0] = world.isend(0, 1, out_msg);
    reqs[1] = world.irecv(0, 0, msg);
    mpi::wait_all(reqs, reqs + 2);
    std::cout << msg << ", ";
  }

  return 0;
}
```

We have replaced calls to the `communicator::send` and `communicator::recv` members with similar calls to their non-blocking counterparts, `communicator::isend` and `communicator::irecv`. The prefix **i** indicates that the operations return immediately with a `mpi::request` object, which allows one to query the status of a communication request (see the `test` method) or wait until it has completed (see the `wait` method). Multiple requests can be completed at the same time with the `wait_all` operation.

If you run this program multiple times, you may see some strange results: namely, some runs will produce:

```
Hello, world!
```

while others will produce:

```
world!
Hello,
```

or even some garbled version of the letters in "Hello" and "world". This indicates that there is some parallelism in the program, because after both messages are (simultaneously) transmitted, both processes will concurrent execute their print statements. For both performance and correctness, non-blocking communication operations are critical to many parallel applications using MPI.

User-defined data types

The inclusion of `boost/serialization/string.hpp` in the previous examples is very important: it makes values of type `std::string` serializable, so that they can be be transmitted using Boost.MPI. In general, built-in C++ types (`int`s, `float`s, characters, etc.) can be transmitted over MPI directly, while user-defined and library-defined types will need to first be serialized (packed) into a format that is amenable to transmission. Boost.MPI relies on the Boost.Serialization library to serialize and deserialize data types.

For types defined by the standard library (such as `std::string` or `std::vector`) and some types in Boost (such as `boost::variant`), the Boost.Serialization library already contains all of the required serialization code. In these cases, you need only include the appropriate header from the `boost/serialization` directory.

For types that do not already have a serialization header, you will first need to implement serialization code before the types can be transmitted using Boost.MPI. Consider a simple class `gps_position` that contains members `degrees`, `minutes`, and `seconds`. This class is made serializable by making it a friend of `boost::serialization::access` and introducing the templated `serialize()` function, as follows:

```
class gps_position
{
private:
    friend class boost::serialization::access;

    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        ar & degrees;
        ar & minutes;
        ar & seconds;
    }

    int degrees;
    int minutes;
    float seconds;
public:
    gps_position(){};
    gps_position(int d, int m, float s) :
        degrees(d), minutes(m), seconds(s)
    {}
};
```

Complete information about making types serializable is beyond the scope of this tutorial. For more information, please see the Boost.Serialization library tutorial from which the above example was extracted. One important side benefit of making types serializable for Boost.MPI is that they become serializable for any other usage, such as storing the objects to disk to manipulated them in XML.

Some serializable types, like `gps_position` above, have a fixed amount of data stored at fixed field positions. When this is the case, Boost.MPI can optimize their serialization and transmission to avoid extraneous copy operations. To enable this optimization, users should specialize the type trait `is_mpi_datatype`, e.g.:

```
namespace boost { namespace mpi {
  template <>
  struct is_mpi_datatype<gps_position> : mpl::true_ { };
} }
```

For non-template types we have defined a macro to simplify declaring a type as an MPI datatype

```
BOOST_IS_MPI_DATATYPE(gps_position)
```

For composite traits, the specialization of `is_mpi_datatype` may depend on `is_mpi_datatype` itself. For instance, a `boost::array` object is fixed only when the type of the parameter it stores is fixed:

```
namespace boost { namespace mpi {
  template <typename T, std::size_t N>
  struct is_mpi_datatype<array<T, N> >
    : public is_mpi_datatype<T> { };
} }
```

The redundant copy elimination optimization can only be applied when the shape of the data type is completely fixed. Variable-length types (e.g., strings, linked lists) and types that store pointers cannot use the optimiation, but Boost.MPI will be unable to detect this error at compile time. Attempting to perform this optimization when it is not correct will likely result in segmentation faults and other strange program behavior.

Boost.MPI can transmit any user-defined data type from one process to another. Built-in types can be transmitted without any extra effort; library-defined types require the inclusion of a serialization header; and user-defined types will require the addition of serialization code. Fixed data types can be optimized for transmission using the `is_mpi_datatype` type trait.

Collective operations

Point-to-point operations are the core message passing primitives in Boost.MPI. However, many message-passing applications also require higher-level communication algorithms that combine or summarize the data stored on many different processes. These algorithms support many common tasks such as "broadcast this value to all processes", "compute the sum of the values on all processors" or "find the global minimum."

Broadcast

The `broadcast` algorithm is by far the simplest collective operation. It broadcasts a value from a single process to all other processes within a `communicator`. For instance, the following program broadcasts "Hello, World!" from process 0 to every other process. (`hello_world_broadcast.cpp`)

```
#include <boost/mpi.hpp>
#include <iostream>
#include <string>
#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  std::string value;
  if (world.rank() == 0) {
    value = "Hello, World!";
  }

  broadcast(world, value, 0);

  std::cout << "Process #" << world.rank() << " says " << value
            << std::endl;
  return 0;
}
```

Running this program with seven processes will produce a result such as:

```
Process #0 says Hello, World!
Process #2 says Hello, World!
Process #1 says Hello, World!
Process #4 says Hello, World!
Process #3 says Hello, World!
Process #5 says Hello, World!
Process #6 says Hello, World!
```

Gather

The `gather` collective gathers the values produced by every process in a communicator into a vector of values on the "root" process (specified by an argument to `gather`). The /i/th element in the vector will correspond to the value gathered fro mthe /i/th process. For instance, in the following program each process computes its own random number. All of these random numbers are gathered at process 0 (the "root" in this case), which prints out the values that correspond to each processor. (`random_gather.cpp`)

```
#include <boost/mpi.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  std::srand(time(0) + world.rank());
  int my_number = std::rand();
  if (world.rank() == 0) {
    std::vector<int> all_numbers;
    gather(world, my_number, all_numbers, 0);
    for (int proc = 0; proc < world.size(); ++proc)
      std::cout << "Process #" << proc << " thought of "
                << all_numbers[proc] << std::endl;
  } else {
    gather(world, my_number, 0);
  }

  return 0;
}
```

Executing this program with seven processes will result in output such as the following. Although the random values will change from one run to the next, the order of the processes in the output will remain the same because only process 0 writes to `std::cout`.

```
Process #0 thought of 332199874
Process #1 thought of 20145617
Process #2 thought of 1862420122
Process #3 thought of 480422940
Process #4 thought of 1253380219
Process #5 thought of 949458815
Process #6 thought of 650073868
```

The `gather` operation collects values from every process into a vector at one process. If instead the values from every process need to be collected into identical vectors on every process, use the `all_gather` algorithm, which is semantically equivalent to calling `gather` followed by a `broadcast` of the resulting vector.

Reduce

The `reduce` collective summarizes the values from each process into a single value at the user-specified "root" process. The Boost.MPI `reduce` operation is similar in spirit to the STL `accumulate` operation, because it takes a sequence of values (one per process) and combines them via a function object. For instance, we can randomly generate values in each process and the compute the minimum value over all processes via a call to `reduce` (`random_min.cpp`)::

```
#include <boost/mpi.hpp>
#include <iostream>
#include <cstdlib>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  std::srand(time(0) + world.rank());
  int my_number = std::rand();

  if (world.rank() == 0) {
    int minimum;
    reduce(world, my_number, minimum, mpi::minimum<int>(), 0);
    std::cout << "The minimum value is " << minimum << std::endl;
  } else {
    reduce(world, my_number, mpi::minimum<int>(), 0);
  }

  return 0;
}
```

The use of `mpi::minimum<int>` indicates that the minimum value should be computed. `mpi::minimum<int>` is a binary function object that compares its two parameters via `<` and returns the smaller value. Any associative binary function or function object will work. For instance, to concatenate strings with `reduce` one could use the function object `std::plus<std::string>` (`string_cat.cpp`):

```
#include <boost/mpi.hpp>
#include <iostream>
#include <string>
#include <functional>
#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  std::string names[10] = { "zero ", "one ", "two ", "three ",
                            "four ", "five ", "six ", "seven ",
                            "eight ", "nine " };

  std::string result;
  reduce(world,
         world.rank() < 10? names[world.rank()]
                          : std::string("many "),
         result, std::plus<std::string>(), 0);

  if (world.rank() == 0)
    std::cout << "The result is " << result << std::endl;

  return 0;
}
```

In this example, we compute a string for each process and then perform a reduction that concatenates all of the strings together into one, long string. Executing this program with seven processors yields the following output:

```
The result is zero one two three four five six
```

Any kind of binary function objects can be used with `reduce`. For instance, and there are many such function objects in the C++ standard `<functional>` header and the Boost.MPI header `<boost/mpi/operations.hpp>`. Or, you can create your own function object. Function objects used with `reduce` must be associative, i.e. `f(x, f(y, z))` must be equivalent to `f(f(x, y), z)`. If they are also commutative (i..e, `f(x, y) == f(y, x)`), Boost.MPI can use a more efficient implementation of `reduce`. To state that a function object is commutative, you will need to specialize the class `is_commutative`. For instance, we could modify the previous example by telling Boost.MPI that string concatenation is commutative:

```
namespace boost { namespace mpi {

  template<>
  struct is_commutative<std::plus<std::string>, std::string>
    : mpl::true_ { };

} } // end namespace boost::mpi
```

By adding this code prior to `main()`, Boost.MPI will assume that string concatenation is commutative and employ a different parallel algorithm for the `reduce` operation. Using this algorithm, the program outputs the following when run with seven processes:

```
The result is zero one four five six two three
```

Note how the numbers in the resulting string are in a different order: this is a direct result of Boost.MPI reordering operations. The result in this case differed from the non-commutative result because string concatenation is not commutative: `f("x", "y")` is not the same as `f("y", "x")`, because argument order matters. For truly commutative operations (e.g., integer addition), the more efficient commutative algorithm will produce the same result as the non-commutative algorithm. Boost.MPI also performs direct mappings from function objects in `<functional>` to `MPI_Op` values predefined by MPI (e.g., `MPI_SUM`, `MPI_MAX`); if you have your own function objects that can take advantage of this mapping, see the class template `is_mpi_op`.

Like `gather`, `reduce` has an "all" variant called `all_reduce` that performs the reduction operation and broadcasts the result to all processes. This variant is useful, for instance, in establishing global minimum or maximum values.

Managing communicators

Communication with Boost.MPI always occurs over a communicator. A communicator contains a set of processes that can send messages among themselves and perform collective operations. There can be many communicators within a single program, each of which contains its own isolated communication space that acts independently of the other communicators.

When the MPI environment is initialized, only the "world" communicator (called `MPI_COMM_WORLD` in the MPI C and Fortran bindings) is available. The "world" communicator, accessed by default-constructing a `mpi::communicator` object, contains all of the MPI processes present when the program begins execution. Other communicators can then be constructed by duplicating or building subsets of the "world" communicator. For instance, in the following program we split the processes into two groups: one for processes generating data and the other for processes that will collect the data. (`generate_collect.cpp`)

```
#include <boost/mpi.hpp>
#include <iostream>
#include <cstdlib>
#include <boost/serialization/vector.hpp>
namespace mpi = boost::mpi;

enum message_tags {msg_data_packet, msg_broadcast_data, msg_finished};

void generate_data(mpi::communicator local, mpi::communicator world);
void collect_data(mpi::communicator local, mpi::communicator world);

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  bool is_generator = world.rank() < 2 * world.size() / 3;
  mpi::communicator local = world.split(is_generator? 0 : 1);
  if (is_generator) generate_data(local, world);
  else collect_data(local, world);

  return 0;
}
```

When communicators are split in this way, their processes retain membership in both the original communicator (which is not altered by the split) and the new communicator. However, the ranks of the processes may be different from one communicator to the next, because the rank values within a communicator are always contiguous values starting at zero. In the example above, the first two thirds of the processes become "generators" and the remaining processes become "collectors". The ranks of the "collectors" in the `world` communicator will be 2/3 `world.size()` and greater, whereas the ranks of the same collector processes in the `local` communicator will start at zero. The following excerpt from `collect_data()` (in `generate_collect.cpp`) illustrates how to manage multiple communicators:

```
mpi::status msg = world.probe();
if (msg.tag() == msg_data_packet) {
  // Receive the packet of data
  std::vector<int> data;
  world.recv(msg.source(), msg.tag(), data);

  // Tell each of the collectors that we'll be broadcasting some data
  for (int dest = 1; dest < local.size(); ++dest)
    local.send(dest, msg_broadcast_data, msg.source());

  // Broadcast the actual data.
  broadcast(local, data, 0);
}
```

The code in this except is executed by the "master" collector, e.g., the node with rank 2/3 `world.size()` in the `world` communicator and rank 0 in the `local` (collector) communicator. It receives a message from a generator via the `world` communicator, then broadcasts the message to each of the collectors via the `local` communicator.

For more control in the creation of communicators for subgroups of processes, the Boost.MPI `group` provides facilities to compute the union (`|`), intersection (`&`), and difference (`-`) of two groups, generate arbitrary subgroups, etc.

Separating structure from content

When communicating data types over MPI that are not fundamental to MPI (such as strings, lists, and user-defined data types), Boost.MPI must first serialize these data types into a buffer and then communicate them; the receiver then copies the results into a buffer before deserializing into an object on the other end. For some data types, this overhead can be eliminated by using `is_mpi_datatype`. However, variable-length data types such as strings and lists cannot be MPI data types.

Boost.MPI supports a second technique for improving performance by separating the structure of these variable-length data structures from the content stored in the data structures. This feature is only beneficial when the shape of the data structure remains the same but the content of the data structure will need to be communicated several times. For instance, in a finite element analysis the structure of the mesh may be fixed at the beginning of computation but the various variables on the cells of the mesh (temperature, stress, etc.) will be communicated many times within the iterative analysis process. In this case, Boost.MPI allows one to first send the "skeleton" of the mesh once, then transmit the "content" multiple times. Since the content need not contain any information about the structure of the data type, it can be transmitted without creating separate communication buffers.

To illustrate the use of skeletons and content, we will take a somewhat more limited example wherein a master process generates random number sequences into a list and transmits them to several slave processes. The length of the list will be fixed at program startup, so the content of the list (i.e., the current sequence of numbers) can be transmitted efficiently. The complete example is available in `example/random_content.cpp`. We being with the master process (rank 0), which builds a list, communicates its structure via a `skeleton`, then repeatedly generates random number sequences to be broadcast to the slave processes via `content`:

```
// Generate the list and broadcast its structure
std::list<int> l(list_len);
broadcast(world, mpi::skeleton(l), 0);

// Generate content several times and broadcast out that content
mpi::content c = mpi::get_content(l);
for (int i = 0; i < iterations; ++i) {
  // Generate new random values
  std::generate(l.begin(), l.end(), &random);

  // Broadcast the new content of l
  broadcast(world, c, 0);
}

// Notify the slaves that we're done by sending all zeroes
std::fill(l.begin(), l.end(), 0);
broadcast(world, c, 0);
```

The slave processes have a very similar structure to the master. They receive (via the `broadcast()` call) the skeleton of the data structure, then use it to build their own lists of integers. In each iteration, they receive via another `broadcast()` the new content in the data structure and compute some property of the data:

```
// Receive the content and build up our own list
std::list<int> l;
broadcast(world, mpi::skeleton(l), 0);

mpi::content c = mpi::get_content(l);
int i = 0;
do {
  broadcast(world, c, 0);

  if (std::find_if
       (l.begin(), l.end(),
        std::bind1st(std::not_equal_to<int>(), 0)) == l.end())
    break;

  // Compute some property of the data.

  ++i;
} while (true);
```

The skeletons and content of any Serializable data type can be transmitted either via the `send` and `recv` members of the `communicator` class (for point-to-point communicators) or broadcast via the `broadcast()` collective. When separating a data structure into a skeleton and content, be careful not to modify the data structure (either on the sender side or the receiver side) without transmitting the skeleton again. Boost.MPI can not detect these accidental modifications to the data structure, which will likely result in incorrect data being transmitted or unstable programs.

Performance optimizations

Serialization optimizations

To obtain optimal performance for small fixed-length data types not containing any pointers it is very important to mark them using the type traits of Boost.MPI and Boost.Serialization.

It was alredy discussed that fixed length types containing no pointers can be using as `is_mpi_datatype`, e.g.:

```
namespace boost { namespace mpi {
  template <>
  struct is_mpi_datatype<gps_position> : mpl::true_ { };
} }
```

or the equivalent macro

```
BOOST_IS_MPI_DATATYPE(gps_position)
```

In addition it can give a substantial performance gain to turn off tracking and versioning for these types, if no pointers to these types are used, by using the traits classes or helper macros of Boost.Serialization:

```
BOOST_CLASS_TRACKING(gps_position,track_never)
BOOST_CLASS_IMPLEMENTATION(gps_position,object_serializable)
```

Homogeneous machines

More optimizations are possible on homogeneous machines, by avoiding MPI_Pack/MPI_Unpack calls but using direct bitwise copy. This feature can be enabled by defining the macro BOOST_MPI_HOMOGENEOUS when building Boost.MPI and when building the application.

In addition all classes need to be marked both as is_mpi_datatype and as is_bitwise_serializable, by using the helper macro of Boost.Serialization:

```
BOOST_IS_BITWISE_SERIALIZABLE(gps_position)
```

Usually it is safe to serialize a class for which is_mpi_datatype is true by using binary copy of the bits. The exception are classes for which some members should be skipped for serialization.

Mapping from C MPI to Boost.MPI

This section provides tables that map from the functions and constants of the standard C MPI to their Boost.MPI equivalents. It will be most useful for users that are already familiar with the C or Fortran interfaces to MPI, or for porting existing parallel programs to Boost.MPI.

Table 1. Point-to-point communication

C Function/Constant	Boost.MPI Equivalent
MPI_ANY_SOURCE	any_source
MPI_ANY_TAG	any_tag
MPI_Bsend	unsupported
MPI_Bsend_init	unsupported
MPI_Buffer_attach	unsupported
MPI_Buffer_detach	unsupported
MPI_Cancel	request::cancel
MPI_Get_count	status::count
MPI_Ibsend	unsupported
MPI_Iprobe	communicator::iprobe
MPI_Irsend	unsupported
MPI_Isend	communicator::isend
MPI_Issend	unsupported
MPI_Irecv	communicator::irecv
MPI_Probe	communicator::probe
MPI_PROC_NULL	unsupported
MPI_Recv	communicator::recv
MPI_Recv_init	unsupported
MPI_Request_free	unsupported
MPI_Rsend	unsupported
MPI_Rsend_init	unsupported
MPI_Send	communicator::send
MPI_Sendrecv	unsupported
MPI_Sendrecv_replace	unsupported
MPI_Send_init	unsupported
MPI_Ssend	unsupported
MPI_Ssend_init	unsupported

C Function/Constant	Boost.MPI Equivalent
`MPI_Start`	unsupported
`MPI_Startall`	unsupported
`MPI_Test`	`request::test`
`MPI_Testall`	`test_all`
`MPI_Testany`	`test_any`
`MPI_Testsome`	`test_some`
`MPI_Test_cancelled`	`status::cancelled`
`MPI_Wait`	`request::wait`
`MPI_Waitall`	`wait_all`
`MPI_Waitany`	`wait_any`
`MPI_Waitsome`	`wait_some`

Boost.MPI automatically maps C and C++ data types to their MPI equivalents. The following table illustrates the mappings between C++ types and MPI datatype constants.

Table 2. Datatypes

C Constant	Boost.MPI Equivalent
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED_INT`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	unused
`MPI_PACKED`	used internally for serialized data types
`MPI_LONG_LONG_INT`	`long long int`, if supported by compiler
`MPI_UNSIGNED_LONG_LONG_INT`	`unsigned long long int`, if supported by compiler
`MPI_FLOAT_INT`	`std::pair<float, int>`
`MPI_DOUBLE_INT`	`std::pair<double, int>`
`MPI_LONG_INT`	`std::pair<long, int>`
`MPI_2INT`	`std::pair<int, int>`
`MPI_SHORT_INT`	`std::pair<short, int>`
`MPI_LONG_DOUBLE_INT`	`std::pair<long double, int>`

Boost.MPI does not provide direct wrappers to the MPI derived datatypes functionality. Instead, Boost.MPI relies on the Boost.Serialization library to construct MPI datatypes for user-defined classe. The section on user-defined data types describes this mechanism, which is used for types that marked as "MPI datatypes" using `is_mpi_datatype`.

The derived datatypes table that follows describes which C++ types correspond to the functionality of the C MPI's datatype constructor. Boost.MPI may not actually use the C MPI function listed when building datatypes of a certain form. Since the actual datatypes built by Boost.MPI are typically hidden from the user, many of these operations are called internally by Boost.MPI.

Table 3. Derived datatypes

C Function/Constant	Boost.MPI Equivalent
`MPI_Address`	used automatically in Boost.MPI
`MPI_Type_commit`	used automatically in Boost.MPI
`MPI_Type_contiguous`	arrays
`MPI_Type_extent`	used automatically in Boost.MPI
`MPI_Type_free`	used automatically in Boost.MPI
`MPI_Type_hindexed`	any type used as a subobject
`MPI_Type_hvector`	unused
`MPI_Type_indexed`	any type used as a subobject
`MPI_Type_lb`	unsupported
`MPI_Type_size`	used automatically in Boost.MPI
`MPI_Type_struct`	user-defined classes and structs
`MPI_Type_ub`	unsupported
`MPI_Type_vector`	used automatically in Boost.MPI

MPI's packing facilities store values into a contiguous buffer, which can later be transmitted via MPI and unpacked into separate values via MPI's unpacking facilities. As with datatypes, Boost.MPI provides an abstract interface to MPI's packing and unpacking facilities. In particular, the two archive classes `packed_oarchive` and `packed_iarchive` can be used to pack or unpack a contiguous buffer using MPI's facilities.

Table 4. Packing and unpacking

C Function	Boost.MPI Equivalent
`MPI_Pack`	`packed_oarchive`
`MPI_Pack_size`	used internally by Boost.MPI
`MPI_Unpack`	`packed_iarchive`

Boost.MPI supports a one-to-one mapping for most of the MPI collectives. For each collective provided by Boost.MPI, the underlying C MPI collective will be invoked when it is possible (and efficient) to do so.

Table 5. Collectives

C Function	Boost.MPI Equivalent
`MPI_Allgather`	`all_gather`
`MPI_Allgatherv`	most uses supported by `all_gather`
`MPI_Allreduce`	`all_reduce`
`MPI_Alltoall`	`all_to_all`
`MPI_Alltoallv`	most uses supported by `all_to_all`
`MPI_Barrier`	`communicator::barrier`
`MPI_Bcast`	`broadcast`
`MPI_Gather`	`gather`
`MPI_Gatherv`	most uses supported by `gather`
`MPI_Reduce`	`reduce`
`MPI_Reduce_scatter`	unsupported
`MPI_Scan`	`scan`
`MPI_Scatter`	`scatter`
`MPI_Scatterv`	most uses supported by `scatter`

Boost.MPI uses function objects to specify how reductions should occur in its equivalents to `MPI_Allreduce`, `MPI_Reduce`, and `MPI_Scan`. The following table illustrates how predefined and user-defined reduction operations can be mapped between the C MPI and Boost.MPI.

Table 6. Reduction operations

C Constant	Boost.MPI Equivalent
`MPI_BAND`	`bitwise_and`
`MPI_BOR`	`bitwise_or`
`MPI_BXOR`	`bitwise_xor`
`MPI_LAND`	`std::logical_and`
`MPI_LOR`	`std::logical_or`
`MPI_LXOR`	`logical_xor`
`MPI_MAX`	`maximum`
`MPI_MAXLOC`	unsupported
`MPI_MIN`	`minimum`
`MPI_MINLOC`	unsupported
`MPI_Op_create`	used internally by Boost.MPI
`MPI_Op_free`	used internally by Boost.MPI
`MPI_PROD`	`std::multiplies`
`MPI_SUM`	`std::plus`

MPI defines several special communicators, including `MPI_COMM_WORLD` (including all processes that the local process can communicate with), `MPI_COMM_SELF` (including only the local process), and `MPI_COMM_EMPTY` (including no processes). These special communicators are all instances of the `communicator` class in Boost.MPI.

Table 7. Predefined communicators

C Constant	Boost.MPI Equivalent
`MPI_COMM_WORLD`	a default-constructed `communicator`
`MPI_COMM_SELF`	a `communicator` that contains only the current process
`MPI_COMM_EMPTY`	a `communicator` that evaluates false

Boost.MPI supports groups of processes through its `group` class.

Table 8. Group operations and constants

C Function/Constant	Boost.MPI Equivalent
`MPI_GROUP_EMPTY`	a default-constructed `group`
`MPI_Group_size`	`group::size`
`MPI_Group_rank`	memberref boost::mpi::group::rank `group::rank`
`MPI_Group_translate_ranks`	memberref boost::mpi::group::translate_ranks `group::translate_ranks`
`MPI_Group_compare`	operators == and `!=`
`MPI_IDENT`	operators == and `!=`
`MPI_SIMILAR`	operators == and `!=`
`MPI_UNEQUAL`	operators == and `!=`
`MPI_Comm_group`	`communicator::group`
`MPI_Group_union`	operator `|` for groups
`MPI_Group_intersection`	operator & for groups
`MPI_Group_difference`	operator `-` for groups
`MPI_Group_incl`	`group::include`
`MPI_Group_excl`	`group::exclude`
`MPI_Group_range_incl`	unsupported
`MPI_Group_range_excl`	unsupported
`MPI_Group_free`	used automatically in Boost.MPI

Boost.MPI provides manipulation of communicators through the `communicator` class.

Table 9. Communicator operations

C Function	Boost.MPI Equivalent
`MPI_Comm_size`	`communicator::size`
`MPI_Comm_rank`	`communicator::rank`
`MPI_Comm_compare`	operators == and `!=`
`MPI_Comm_dup`	`communicator` class constructor using `comm_duplicate`
`MPI_Comm_create`	`communicator` constructor
`MPI_Comm_split`	`communicator::split`
`MPI_Comm_free`	used automatically in Boost.MPI

Boost.MPI currently provides support for inter-communicators via the `intercommunicator` class.

Table 10. Inter-communicator operations

C Function	Boost.MPI Equivalent
`MPI_Comm_test_inter`	use `communicator::as_intercommunicator`
`MPI_Comm_remote_size`	`boost::mpi::intercommunicator::remote_size` `intercommunicator::remote_size`
`MPI_Comm_remote_group`	`intercommunicator::remote_group`
`MPI_Intercomm_create`	`intercommunicator` constructor
`MPI_Intercomm_merge`	`intercommunicator::merge`

Boost.MPI currently provides no support for attribute caching.

Table 11. Attributes and caching

C Function/Constant	Boost.MPI Equivalent
`MPI_NULL_COPY_FN`	unsupported
`MPI_NULL_DELETE_FN`	unsupported
`MPI_KEYVAL_INVALID`	unsupported
`MPI_Keyval_create`	unsupported
`MPI_Copy_function`	unsupported
`MPI_Delete_function`	unsupported
`MPI_Keyval_free`	unsupported
`MPI_Attr_put`	unsupported
`MPI_Attr_get`	unsupported
`MPI_Attr_delete`	unsupported

Boost.MPI will provide complete support for creating communicators with different topologies and later querying those topologies. Support for graph topologies is provided via an interface to the Boost Graph Library (BGL), where a communicator can be created which matches the structure of any BGL graph, and the graph topology of a communicator can be viewed as a BGL graph for use in existing, generic graph algorithms.

Table 12. Process topologies

C Function/Constant	Boost.MPI Equivalent
`MPI_GRAPH`	unnecessary; use `communicator::has_graph_topology`
`MPI_CART`	unnecessary; use `communicator::has_cartesian_topology`
`MPI_Cart_create`	unsupported
`MPI_Dims_create`	unsupported
`MPI_Graph_create`	`communicator::with_graph_topology`
`MPI_Topo_test`	`communicator::has_graph_topology`, `communicator::has_cartesian_topology`
`MPI_Graphdims_get`	`num_vertices`, `num_edges`
`MPI_Graph_get`	`vertices`, `edges`
`MPI_Cartdim_get`	unsupported
`MPI_Cart_get`	unsupported
`MPI_Cart_rank`	unsupported
`MPI_Cart_coords`	unsupported
`MPI_Graph_neighbors_count`	`out_degree`
`MPI_Graph_neighbors`	`out_edges`, `adjacent_vertices`
`MPI_Cart_shift`	unsupported
`MPI_Cart_sub`	unsupported
`MPI_Cart_map`	unsupported
`MPI_Graph_map`	unsupported

Boost.MPI supports environmental inquires through the `environment` class.

Table 13. Environmental inquiries

C Function/Constant	Boost.MPI Equivalent
`MPI_TAG_UB`	unnecessary; use `environment::max_tag`
`MPI_HOST`	unnecessary; use `environment::host_rank`
`MPI_IO`	unnecessary; use `environment::io_rank`
`MPI_Get_processor_name`	`environment::processor_name`

Boost.MPI translates MPI errors into exceptions, reported via the `exception` class.

Table 14. Error handling

C Function/Constant	Boost.MPI Equivalent
`MPI_ERRORS_ARE_FATAL`	unused; errors are translated into Boost.MPI exceptions
`MPI_ERRORS_RETURN`	unused; errors are translated into Boost.MPI exceptions
`MPI_errhandler_create`	unused; errors are translated into Boost.MPI exceptions
`MPI_errhandler_set`	unused; errors are translated into Boost.MPI exceptions
`MPI_errhandler_get`	unused; errors are translated into Boost.MPI exceptions
`MPI_errhandler_free`	unused; errors are translated into Boost.MPI exceptions
`MPI_Error_string`	used internally by Boost.MPI
`MPI_Error_class`	`exception::error_class`

The MPI timing facilities are exposed via the Boost.MPI `timer` class, which provides an interface compatible with the Boost Timer library.

Table 15. Timing facilities

C Function/Constant	Boost.MPI Equivalent
`MPI_WTIME_IS_GLOBAL`	unnecessary; use `timer::time_is_global`
`MPI_Wtime`	use `timer::elapsed` to determine the time elapsed from some specific starting point
`MPI_Wtick`	`timer::elapsed_min`

MPI startup and shutdown are managed by the construction and descruction of the Boost.MPI `environment` class.

Table 16. Startup/shutdown facilities

C Function	Boost.MPI Equivalent
`MPI_Init`	`environment` constructor
`MPI_Finalize`	`environment` destructor
`MPI_Initialized`	`environment::initialized`
`MPI_Abort`	`environment::abort`

Boost.MPI does not provide any support for the profiling facilities in MPI 1.1.

Table 17. Profiling interface

C Function	Boost.MPI Equivalent
`PMPI_*` routines	unsupported
`MPI_Pcontrol`	unsupported

Reference

Header <boost/mpi.hpp>

This file is a top-level convenience header that includes all of the Boost.MPI library headers. Users concerned about compile time may wish to include only specific headers from the Boost.MPI library.

Header <boost/mpi/allocator.hpp>

This header provides an STL-compliant allocator that uses the MPI-2 memory allocation facilities.

```
namespace boost {
  namespace mpi {
    template<> class allocator<void>;

    template<typename T> class allocator;
    template<typename T1, typename T2>
      bool operator==(const allocator< T1 > &,
                      const allocator< T2 > &);
    template<typename T1, typename T2>
      bool operator!=(const allocator< T1 > &,
                      const allocator< T2 > &);
  }
}
```

Class allocator<void>

boost::mpi::allocator<void> — Allocator specialization for `void` value types.

Synopsis

```
// In header: <boost/mpi/allocator.hpp>


class allocator<void> {
public:
  // types
  typedef void *       pointer;
  typedef const void * const_pointer;
  typedef void         value_type;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef allocator< U > other;
  };
};
```

Description

The `void` specialization of `allocator` is useful only for rebinding to another, different value type.

Struct template rebind

boost::mpi::allocator<void>::rebind

Synopsis

```
// In header: <boost/mpi/allocator.hpp>


template<typename U>
struct rebind {
  // types
  typedef allocator< U > other;
};
```

Class template allocator

boost::mpi::allocator — Standard Library-compliant allocator for the MPI-2 memory allocation routines.

Synopsis

```
// In header: <boost/mpi/allocator.hpp>

template<typename T>
class allocator {
public:
  // types
  typedef std::size_t    size_type;        // Holds the size of objects.
  typedef std::ptrdiff_t difference_type;  // Holds the number of elements between two pointers.
  typedef T *            pointer;          // A pointer to an object of type T.
  typedef const T *      const_pointer;    // A pointer to a constant object of type T.
  typedef T &            reference;        // A reference to an object of type T.
  typedef const T &      const_reference;  // A reference to a constant object of type T.
  typedef T              value_type;       // The type of memory allocated by this allocator.

  // member classes/structs/unions

  // Retrieve the type of an allocator similar to this allocator
  // but for a different value type.
  template<typename U>
  struct rebind {
    // types
    typedef allocator< U > other;
  };

  // construct/copy/destruct
  allocator();
  allocator(const allocator &);
  template<typename U> allocator(const allocator< U > &);
  ~allocator();

  // public member functions
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  pointer allocate(size_type,
                   allocator< void >::const_pointer = 0);
  void deallocate(pointer, size_type);
  size_type max_size() const;
  void construct(pointer, const T &);
  void destroy(pointer);
};
```

Description

This allocator provides a standard C++ interface to the `MPI_Alloc_mem` and `MPI_Free_mem` routines of MPI-2. It is intended to be used with the containers in the Standard Library (`vector`, in particular) in cases where the contents of the container will be directly transmitted via MPI. This allocator is also used internally by the library for character buffers that will be used in the transmission of data.

The `allocator` class template only provides MPI memory allocation when the underlying MPI implementation is either MPI-2 compliant or is known to provide `MPI_Alloc_mem` and `MPI_Free_mem` as extensions. When the MPI memory allocation routines are not available, `allocator` is brought in directly from namespace `std`, so that standard allocators are used throughout. The macro `BOOST_MPI_HAS_MEMORY_ALLOCATION` will be defined when the MPI-2 memory allocation facilities are available.

`allocator` public construct/copy/destruct

1. ```
   allocator();
   ```

   Default-construct an allocator.

2. ```
   allocator(const allocator &);
   ```

   Copy-construct an allocator.

3. ```
   template<typename U> allocator(const allocator< U > &);
   ```

   Copy-construct an allocator from another allocator for a different value type.

4. ```
   ~allocator();
   ```

   Destroy an allocator.

`allocator` public member functions

1. ```
   pointer address(reference x) const;
   ```

   Returns the address of object `x`.

2. ```
   const_pointer address(const_reference x) const;
   ```

   Returns the address of object `x`.

3. ```
   pointer allocate(size_type n,
                    allocator< void >::const_pointer = 0);
   ```

   Allocate enough memory for `n` elements of type `T`.

   Parameters: `n` The number of elements for which memory should be allocated.
   Returns: a pointer to the newly-allocated memory

4. ```
   void deallocate(pointer p, size_type);
   ```

   Deallocate memory referred to by the pointer `p`.

   Parameters: `p` The pointer whose memory should be deallocated. This pointer shall have been returned from the `allocate()` function and not have already been freed.

5. ```
   size_type max_size() const;
   ```

   Returns the maximum number of elements that can be allocated with `allocate()`.

6. ```
   void construct(pointer p, const T & val);
   ```

   Construct a copy of `val` at the location referenced by `p`.

7.
```
void destroy(pointer p);
```

Destroy the object referenced by `p`.

Struct template rebind

boost::mpi::allocator::rebind — Retrieve the type of an allocator similar to this allocator but for a different value type.

Synopsis

```
// In header: <boost/mpi/allocator.hpp>



// Retrieve the type of an allocator similar to this allocator but
// for a different value type.
template<typename U>
struct rebind {
  // types
  typedef allocator< U > other;
};
```

Specializations

- Class allocator<void>

Function template operator==

boost::mpi::operator== — Compare two allocators for equality.

Synopsis

```
// In header: <boost/mpi/allocator.hpp>


template<typename T1, typename T2>
  bool operator==(const allocator< T1 > &,
                  const allocator< T2 > &);
```

Description

Since MPI allocators have no state, all MPI allocators are equal.

Returns: `true`

Function template operator!=

boost::mpi::operator!= — Compare two allocators for inequality.

Synopsis

```
// In header: <boost/mpi/allocator.hpp>


template<typename T1, typename T2>
  bool operator!=(const allocator< T1 > &,
                  const allocator< T2 > &);
```

Description

Since MPI allocators have no state, all MPI allocators are equal.

Returns: `false`

Header <boost/mpi/collectives.hpp>

This header contains MPI collective operations, which implement various parallel algorithms that require the coordination of all processes within a communicator. The header `collectives_fwd.hpp` provides forward declarations for each of these operations. To include only specific collective algorithms, use the headers `boost/mpi/collectives/algorithm_name.hpp`.

```
namespace boost {
  namespace mpi {
    template<typename T>
      void all_gather(const communicator &, const T &,
                      std::vector< T > &);
    template<typename T>
      void all_gather(const communicator &, const T &, T *);
    template<typename T>
      void all_gather(const communicator &, const T *, int,
                      std::vector< T > &);
    template<typename T>
      void all_gather(const communicator &, const T *, int, T *);
    template<typename T, typename Op>
      void all_reduce(const communicator &, const T &, T &, Op);
    template<typename T, typename Op>
      T all_reduce(const communicator &, const T &, Op);
    template<typename T, typename Op>
      void all_reduce(const communicator &, const T *, int, T *,
                      Op);
    template<typename T>
      void all_to_all(const communicator &,
                      const std::vector< T > &,
                      std::vector< T > &);
    template<typename T>
      void all_to_all(const communicator &, const T *, T *);
    template<typename T>
      void all_to_all(const communicator &,
                      const std::vector< T > &, int,
                      std::vector< T > &);
    template<typename T>
      void all_to_all(const communicator &, const T *, int, T *);
    template<typename T>
      void broadcast(const communicator &, T &, int);
    template<typename T>
      void broadcast(const communicator &, T *, int, int);
    template<typename T>
      void broadcast(const communicator &, skeleton_proxy< T > &,
                     int);
    template<typename T>
      void broadcast(const communicator &,
                     const skeleton_proxy< T > &, int);
    template<typename T>
      void gather(const communicator &, const T &,
                  std::vector< T > &, int);
    template<typename T>
      void gather(const communicator &, const T &, T *, int);
    template<typename T>
      void gather(const communicator &, const T &, int);
    template<typename T>
      void gather(const communicator &, const T *, int,
                  std::vector< T > &, int);
    template<typename T>
      void gather(const communicator &, const T *, int, T *, int);
    template<typename T>
      void gather(const communicator &, const T *, int, int);
    template<typename T>
      void scatter(const communicator &, const std::vector< T > &,
                   T &, int);
    template<typename T>
      void scatter(const communicator &, const T *, T &, int);
    template<typename T>
      void scatter(const communicator &, T &, int);
    template<typename T>
```

```
    void scatter(const communicator &, const std::vector< T > &,
                 T *, int, int);
  template<typename T>
    void scatter(const communicator &, const T *, T *, int, int);
  template<typename T>
    void scatter(const communicator &, T *, int, int);
  template<typename T, typename Op>
    void reduce(const communicator &, const T &, T &, Op, int);
  template<typename T, typename Op>
    void reduce(const communicator &, const T &, Op, int);
  template<typename T, typename Op>
    void reduce(const communicator &, const T *, int, T *, Op,
                int);
  template<typename T, typename Op>
    void reduce(const communicator &, const T *, int, Op, int);
  template<typename T, typename Op>
    void scan(const communicator &, const T &, T &, Op);
  template<typename T, typename Op>
    T scan(const communicator &, const T &, Op);
  template<typename T, typename Op>
    void scan(const communicator &, const T *, int, T *, Op);
  }
}
```

Function all_gather

boost::mpi::all_gather — Gather the values stored at every process into vectors of values from each process.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T>
  void all_gather(const communicator & comm, const T & in_value,
                  std::vector< T > & out_values);
template<typename T>
  void all_gather(const communicator & comm, const T & in_value,
                  T * out_values);
template<typename T>
  void all_gather(const communicator & comm, const T * in_values,
                  int n, std::vector< T > & out_values);
template<typename T>
  void all_gather(const communicator & comm, const T * in_values,
                  int n, T * out_values);
```

Description

`all_gather` is a collective algorithm that collects the values stored at each process into a vector of values indexed by the process number they came from. The type `T` of the values may be any type that is serializable or has an associated MPI data type.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Allgather` to gather the values.

Parameters:	`comm`	The communicator over which the all-gather will occur.
	`in_value`	The value to be transmitted by each process. To gather an array of values, `in_values` points to the `n` local values to be transmitted.
	`out_values`	A vector or pointer to storage that will be populated with the values from each process, indexed by the process ID number. If it is a vector, the vector will be resized accordingly.

Function all_reduce

boost::mpi::all_reduce — Combine the values stored by each process into a single value available to all processes.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T, typename Op>
  void all_reduce(const communicator & comm, const T & in_value,
                  T & out_value, Op op);
template<typename T, typename Op>
  T all_reduce(const communicator & comm, const T & in_value,
               Op op);
template<typename T, typename Op>
  void all_reduce(const communicator & comm, const T * in_values,
                  int n, T * out_values, Op op);
```

Description

`all_reduce` is a collective algorithm that combines the values stored by each process into a single value available to all processes. The values are combined in a user-defined way, specified via a function object. The type `T` of the values may be any type that is serializable or has an associated MPI data type. One can think of this operation as a `all_gather`, followed by an `std::accumulate()` over the gather values and using the operation `op`.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Allreduce` to perform the reduction. If possible, built-in MPI operations will be used; otherwise, `all_reduce()` will create a custom MPI_Op for the call to MPI_Allreduce.

Parameters:	`comm`	The communicator over which the reduction will occur.
	`in_value`	The local value to be combined with the local values of every other process. For reducing arrays, `in_values` is a pointer to the local values to be reduced and `n` is the number of values to reduce. See `reduce` for more information.
	`op`	The binary operation that combines two values of type `T` and returns a third value of type `T`. For types `T` that has ssociated MPI data types, `op` will either be translated into an `MPI_Op` (via `MPI_Op_create`) or, if possible, mapped directly to a built-in MPI operation. See `is_mpi_op` in the `operations.hpp` header for more details on this mapping. For any non-built-in operation, commutativity will be determined by the `is_commmutative` trait (also in `operations.hpp`): users are encouraged to mark commutative operations as such, because it gives the implementation additional lattitude to optimize the reduction operation.
	`out_value`	Will receive the result of the reduction operation. If this parameter is omitted, the outgoing value will instead be returned.
Returns:	If no `out_value` parameter is supplied, returns the result of the reduction operation.	

Function all_to_all

boost::mpi::all_to_all — Send data from every process to every other process.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T>
  void all_to_all(const communicator & comm,
                  const std::vector< T > & in_values,
                  std::vector< T > & out_values);
template<typename T>
  void all_to_all(const communicator & comm, const T * in_values,
                  T * out_values);
template<typename T>
  void all_to_all(const communicator & comm,
                  const std::vector< T > & in_values, int n,
                  std::vector< T > & out_values);
template<typename T>
  void all_to_all(const communicator & comm, const T * in_values,
                  int n, T * out_values);
```

Description

`all_to_all` is a collective algorithm that transmits `p` values from every process to every other process. On process i, jth value of the `in_values` vector is sent to process j and placed in the ith position of the `out_values` vector in process `j`. The type `T` of the values may be any type that is serializable or has an associated MPI data type. If `n` is provided, then arrays of `n` values will be transferred from one process to another.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Alltoall` to scatter the values.

Parameters:	`comm`	The communicator over which the all-to-all communication will occur.
	`in_values`	A vector or pointer to storage that contains the values to send to each process, indexed by the process ID number.
	`out_values`	A vector or pointer to storage that will be updated to contain the values received from other processes. The jth value in `out_values` will come from the procss with rank j.

Function broadcast

boost::mpi::broadcast — Broadcast a value from a root process to all other processes.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T>
  void broadcast(const communicator & comm, T & value, int root);
template<typename T>
  void broadcast(const communicator & comm, T * values, int n,
                 int root);
template<typename T>
  void broadcast(const communicator & comm,
                 skeleton_proxy< T > & value, int root);
template<typename T>
  void broadcast(const communicator & comm,
                 const skeleton_proxy< T > & value, int root);
```

Description

`broadcast` is a collective algorithm that transfers a value from an arbitrary `root` process to every other process that is part of the given communicator. The `broadcast` algorithm can transmit any Serializable value, values that have associated MPI data types, packed archives, skeletons, and the content of skeletons; see the `send` primitive for communicators for a complete list. The type `T` shall be the same for all processes that are a part of the communicator `comm`, unless packed archives are being transferred: with packed archives, the root sends a `packed_oarchive` or `packed_skeleton_oarchive` whereas the other processes receive a `packed_iarchive` or `packed_skeleton_iarchve`, respectively.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Bcast` to perform the broadcast.

Parameters:

`comm`	The communicator over which the broadcast will occur.
`root`	The rank/process ID of the process that will be transmitting the value.
`value`	The value (or values, if `n` is provided) to be transmitted (if the rank of `comm` is equal to `root`) or received (if the rank of `comm` is not equal to `root`). When the `value` is a `skeleton_proxy`, only the skeleton of the object will be broadcast. In this case, the `root` will build a skeleton from the object help in the proxy and all of the non-roots will reshape the objects held in their proxies based on the skeleton sent from the root.

Function gather

boost::mpi::gather — Gather the values stored at every process into a vector at the root process.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T>
  void gather(const communicator & comm, const T & in_value,
              std::vector< T > & out_values, int root);
template<typename T>
  void gather(const communicator & comm, const T & in_value,
              T * out_values, int root);
template<typename T>
  void gather(const communicator & comm, const T & in_value,
              int root);
template<typename T>
  void gather(const communicator & comm, const T * in_values,
              int n, std::vector< T > & out_values, int root);
template<typename T>
  void gather(const communicator & comm, const T * in_values,
              int n, T * out_values, int root);
template<typename T>
  void gather(const communicator & comm, const T * in_values,
              int n, int root);
```

Description

`gather` is a collective algorithm that collects the values stored at each process into a vector of values at the `root` process. This vector is indexed by the process number that the value came from. The type `T` of the values may be any type that is serializable or has an associated MPI data type.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Gather` to gather the values.

Parameters:		
	`comm`	The communicator over which the gather will occur.
	`in_value`	The value to be transmitted by each process. For gathering arrays of values, `in_values` points to storage for `n*comm`.size() values.
	`out_values`	A vector or pointer to storage that will be populated with the values from each process, indexed by the process ID number. If it is a vector, it will be resized accordingly. For non-root processes, this parameter may be omitted. If it is still provided, however, it will be unchanged.
	`root`	The process ID number that will collect the values. This value must be the same on all processes.

Function scatter

boost::mpi::scatter — Scatter the values stored at the root to all processes within the communicator.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T>
  void scatter(const communicator & comm,
               const std::vector< T > & in_values, T & out_value,
               int root);
template<typename T>
  void scatter(const communicator & comm, const T * in_values,
               T & out_value, int root);
template<typename T>
  void scatter(const communicator & comm, T & out_value, int root);
template<typename T>
  void scatter(const communicator & comm,
               const std::vector< T > & in_values,
               T * out_values, int n, int root);
template<typename T>
  void scatter(const communicator & comm, const T * in_values,
               T * out_values, int n, int root);
template<typename T>
  void scatter(const communicator & comm, T * out_values, int n,
               int root);
```

Description

`scatter` is a collective algorithm that scatters the values stored in the `root` process (inside a vector) to all of the processes in the communicator. The vector `out_values` (only significant at the `root`) is indexed by the process number to which the corresponding value will be sent. The type `T` of the values may be any type that is serializable or has an associated MPI data type.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Scatter` to scatter the values.

Parameters:	`comm`	The communicator over which the gather will occur.
	`in_values`	A vector or pointer to storage that will contain the values to send to each process, indexed by the process rank. For non-root processes, this parameter may be omitted. If it is still provided, however, it will be unchanged.
	`out_value`	The value received by each process. When scattering an array of values, `out_values` points to the `n` values that will be received by each process.
	`root`	The process ID number that will scatter the values. This value must be the same on all processes.

Function reduce

boost::mpi::reduce — Combine the values stored by each process into a single value at the root.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T, typename Op>
  void reduce(const communicator & comm, const T & in_value,
              T & out_value, Op op, int root);
template<typename T, typename Op>
  void reduce(const communicator & comm, const T & in_value,
              Op op, int root);
template<typename T, typename Op>
  void reduce(const communicator & comm, const T * in_values,
              int n, T * out_values, Op op, int root);
template<typename T, typename Op>
  void reduce(const communicator & comm, const T * in_values,
              int n, Op op, int root);
```

Description

`reduce` is a collective algorithm that combines the values stored by each process into a single value at the `root`. The values can be combined arbitrarily, specified via a function object. The type `T` of the values may be any type that is serializable or has an associated MPI data type. One can think of this operation as a `gather` to the `root`, followed by an `std::accumulate()` over the gathered values and using the operation `op`.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Reduce` to perform the reduction. If possible, built-in MPI operations will be used; otherwise, `reduce()` will create a custom MPI_Op for the call to MPI_Reduce.

Parameters:

`comm`	The communicator over which the reduction will occur.
`in_value`	The local value to be combined with the local values of every other process. For reducing arrays, `in_values` contains a pointer to the local values. In this case, `n` is the number of values that will be reduced. Reduction occurs independently for each of the `n` values referenced by `in_values`, e.g., calling reduce on an array of `n` values is like calling `reduce` `n` separate times, one for each location in `in_values` and `out_values`.
`op`	The binary operation that combines two values of type `T` into a third value of type `T`. For types `T` that has ssociated MPI data types, `op` will either be translated into an `MPI_Op` (via `MPI_Op_create`) or, if possible, mapped directly to a built-in MPI operation. See `is_mpi_op` in the `operations.hpp` header for more details on this mapping. For any non-built-in operation, commutativity will be determined by the `is_commmutative` trait (also in `operations.hpp`): users are encouraged to mark commutative operations as such, because it gives the implementation additional lattitude to optimize the reduction operation.
`out_value`	Will receive the result of the reduction operation, but only for the `root` process. Non-root processes may omit if parameter; if they choose to supply the parameter, it will be unchanged. For reducing arrays, `out_values` contains a pointer to the storage for the output values.
`root`	The process ID number that will receive the final, combined value. This value must be the same on all processes.

Function scan

boost::mpi::scan — Compute a prefix reduction of values from all processes in the communicator.

Synopsis

```
// In header: <boost/mpi/collectives.hpp>


template<typename T, typename Op>
  void scan(const communicator & comm, const T & in_value,
            T & out_value, Op op);
template<typename T, typename Op>
  T scan(const communicator & comm, const T & in_value, Op op);
template<typename T, typename Op>
  void scan(const communicator & comm, const T * in_values, int n,
            T * out_values, Op op);
```

Description

`scan` is a collective algorithm that combines the values stored by each process with the values of all processes with a smaller rank. The values can be arbitrarily combined, specified via a function object `op`. The type `T` of the values may be any type that is serializable or has an associated MPI data type. One can think of this operation as a `gather` to some process, followed by an `std::prefix_sum()` over the gathered values using the operation `op`. The ith process returns the ith value emitted by `std::prefix_sum()`.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Scan` to perform the reduction. If possible, built-in MPI operations will be used; otherwise, `scan()` will create a custom `MPI_Op` for the call to MPI_Scan.

Parameters:	`comm`	The communicator over which the prefix reduction will occur.
	`in_value`	The local value to be combined with the local values of other processes. For the array variant, the `in_values` parameter points to the `n` local values that will be combined.
	`op`	The binary operation that combines two values of type `T` into a third value of type `T`. For types `T` that has ssociated MPI data types, `op` will either be translated into an `MPI_Op` (via `MPI_Op_create`) or, if possible, mapped directly to a built-in MPI operation. See `is_mpi_op` in the `operations.hpp` header for more details on this mapping. For any non-built-in operation, commutativity will be determined by the `is_commmutative` trait (also in `operations.hpp`).
	`out_value`	If provided, the ith process will receive the value op(in_value[0], op(in_value[1], op(..., in_value[i]) ...)). For the array variant, `out_values` contains a pointer to storage for the `n` output values. The prefix reduction occurs independently for each of the `n` values referenced by `in_values`, e.g., calling scan on an array of `n` values is like calling `scan` `n` separate times, one for each location in `in_values` and `out_values`.
Returns:	If no `out_value` parameter is provided, returns the result of prefix reduction.	

Header <boost/mpi/collectives_fwd.hpp>

This header provides forward declarations for all of the collective operations contained in the header `collectives.hpp`.

Header <boost/mpi/communicator.hpp>

This header defines the `communicator` class, which is the basis of all communication within Boost.MPI, and provides point-to-point communication operations.

```
namespace boost {
  namespace mpi {
    class communicator;

    enum comm_create_kind;

    const int any_source;    // A constant representing "any process.".
    const int any_tag;    // A constant representing "any tag.".
    BOOST_MPI_DECL bool
    operator==(const communicator &, const communicator &);
    bool operator!=(const communicator &, const communicator &);
  }
}
```

Class communicator

boost::mpi::communicator — A communicator that permits communication and synchronization among a set of processes.

Synopsis

```
// In header: <boost/mpi/communicator.hpp>


class communicator {
public:
  // construct/copy/destruct
  communicator();
  communicator(const MPI_Comm &, comm_create_kind);
  communicator(const communicator &, const boost::mpi::group &);

  // public member functions
  int rank() const;
  int size() const;
  boost::mpi::group group() const;
  template<typename T> void send(int, int, const T &) const;
  template<typename T>
    void send(int, int, const skeleton_proxy< T > &) const;
  template<typename T> void send(int, int, const T *, int) const;
  void send(int, int) const;
  template<typename T> status recv(int, int, T &) const;
  template<typename T>
    status recv(int, int, const skeleton_proxy< T > &) const;
  template<typename T>
    status recv(int, int, skeleton_proxy< T > &) const;
  template<typename T> status recv(int, int, T *, int) const;
  status recv(int, int) const;
  template<typename T> request isend(int, int, const T &) const;
  template<typename T>
    request isend(int, int, const skeleton_proxy< T > &) const;
  template<typename T>
    request isend(int, int, const T *, int) const;
  request isend(int, int) const;
  template<typename T> request irecv(int, int, T &) const;
  template<typename T> request irecv(int, int, T *, int) const;
  request irecv(int, int) const;
  status probe(int = any_source, int = any_tag) const;
  optional< status > iprobe(int = any_source, int = any_tag) const;
  void barrier() const;
  operator bool() const;
  operator MPI_Comm() const;
  communicator split(int) const;
  communicator split(int, int) const;
  optional< intercommunicator > as_intercommunicator() const;
  optional< graph_communicator > as_graph_communicator() const;
  bool has_cartesian_topology() const;
  void abort(int) const;
};
```

Description

The `communicator` class abstracts a set of communicating processes in MPI. All of the processes that belong to a certain communicator can determine the size of the communicator, their rank within the communicator, and communicate with any other processes in the communicator.

`communicator` public construct/copy/destruct

1.
```
communicator();
```

Build a new Boost.MPI communicator for `MPI_COMM_WORLD`.

Constructs a Boost.MPI communicator that attaches to `MPI_COMM_WORLD`. This is the equivalent of constructing with (MPI_COMM_WORLD, comm_attach).

2.
```
communicator(const MPI_Comm & comm, comm_create_kind kind);
```

Build a new Boost.MPI communicator based on the MPI communicator `comm`.

`comm` may be any valid MPI communicator. If `comm` is MPI_COMM_NULL, an empty communicator (that cannot be used for communication) is created and the `kind` parameter is ignored. Otherwise, the `kind` parameters determines how the Boost.MPI communicator will be related to `comm`:

- If `kind` is `comm_duplicate`, duplicate `comm` to create a new communicator. This new communicator will be freed when the Boost.MPI communicator (and all copies of it) is destroyed. This option is only permitted if `comm` is a valid MPI intracommunicator or if the underlying MPI implementation supports MPI 2.0 (which supports duplication of intercommunicators).
- If `kind` is `comm_take_ownership`, take ownership of `comm`. It will be freed automatically when all of the Boost.MPI communicators go out of scope. This option must not be used when `comm` is MPI_COMM_WORLD.
- If `kind` is `comm_attach`, this Boost.MPI communicator will reference the existing MPI communicator `comm` but will not free `comm` when the Boost.MPI communicator goes out of scope. This option should only be used when the communicator is managed by the user or MPI library (e.g., MPI_COMM_WORLD).

3.
```
communicator(const communicator & comm,
             const boost::mpi::group & subgroup);
```

Build a new Boost.MPI communicator based on a subgroup of another MPI communicator.

This routine will construct a new communicator containing all of the processes from communicator `comm` that are listed within the group `subgroup`. Equivalent to `MPI_Comm_create`.

Parameters:

`comm`	An MPI communicator.
`subgroup`	A subgroup of the MPI communicator, `comm`, for which we will construct a new communicator.

`communicator` public member functions

1.
```
int rank() const;
```

Determine the rank of the executing process in a communicator.

This routine is equivalent to `MPI_Comm_rank`.

Returns: The rank of the process in the communicator, which will be a value in [0, size())

2.
```
int size() const;
```

Determine the number of processes in a communicator.

This routine is equivalent to `MPI_Comm_size`.

Returns: The number of processes in the communicator.

3.
```
boost::mpi::group group() const;
```

This routine constructs a new group whose members are the processes within this communicator. Equivalent to calling `MPI_Comm_group`.

4.
```
template<typename T>
  void send(int dest, int tag, const T & value) const;
```

Send data to another process.

This routine executes a potentially blocking send with tag `tag` to the process with rank `dest`. It can be received by the destination process with a matching `recv` call.

The given `value` must be suitable for transmission over MPI. There are several classes of types that meet these requirements:

- Types with mappings to MPI data types: If `is_mpi_datatype<T>` is convertible to `mpl::true_`, then `value` will be transmitted using the MPI data type `get_mpi_datatype<T>()`. All primitive C++ data types that have MPI equivalents, e.g., `int`, `float`, `char`, `double`, etc., have built-in mappings to MPI data types. You may turn a Serializable type with fixed structure into an MPI data type by specializing `is_mpi_datatype` for your type.
- Serializable types: Any type that provides the `serialize()` functionality required by the Boost.Serialization library can be transmitted and received.
- Packed archives and skeletons: Data that has been packed into an `mpi::packed_oarchive` or the skeletons of data that have been backed into an `mpi::packed_skeleton_oarchive` can be transmitted, but will be received as `mpi::packed_iarchive` and `mpi::packed_skeleton_iarchive`, respectively, to allow the values (or skeletons) to be extracted by the destination process.
- Content: Content associated with a previously-transmitted skeleton can be transmitted by `send` and received by `recv`. The receiving process may only receive content into the content of a value that has been constructed with the matching skeleton.

For types that have mappings to an MPI data type (including the concent of a type), an invocation of this routine will result in a single MPI_Send call. For variable-length data, e.g., serialized types and packed archives, two messages will be sent via MPI_Send: one containing the length of the data and the second containing the data itself. Note that the transmission mode for variable-length data is an implementation detail that is subject to change.

Parameters:

`dest`	The rank of the remote process to which the data will be sent.
`tag`	The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.
`value`	The value that will be transmitted to the receiver. The type `T` of this value must meet the aforementioned criteria for transmission.

5.
```
template<typename T>
  void send(int dest, int tag, const skeleton_proxy< T > & proxy) const;
```

Send the skeleton of an object.

This routine executes a potentially blocking send with tag `tag` to the process with rank `dest`. It can be received by the destination process with a matching `recv` call. This variation on `send` will be used when a send of a skeleton is explicitly requested via code such as:

```
    comm.send(dest, tag, skeleton(object));
```

The semantics of this routine are equivalent to that of sending a `packed_skeleton_oarchive` storing the skeleton of the `object`.

Parameters:

`dest`	The rank of the remote process to which the skeleton will be sent.
`proxy`	The `skeleton_proxy` containing a reference to the object whose skeleton will be transmitted.

`tag` The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.

6.
```
template<typename T>
  void send(int dest, int tag, const T * values, int n) const;
```

Send an array of values to another process.

This routine executes a potentially blocking send of an array of data with tag `tag` to the process with rank `dest`. It can be received by the destination process with a matching array `recv` call.

If `T` is an MPI datatype, an invocation of this routine will be mapped to a single call to MPI_Send, using the datatype `get_mpi_datatype<T>()`.

Parameters:
- `dest` The process rank of the remote process to which the data will be sent.
- `n` The number of values stored in the array. The destination process must call receive with at least this many elements to correctly receive the message.
- `tag` The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.
- `values` The array of values that will be transmitted to the receiver. The type `T` of these values must be mapped to an MPI data type.

7.
```
void send(int dest, int tag) const;
```

Send a message to another process without any data.

This routine executes a potentially blocking send of a message to another process. The message contains no extra data, and can therefore only be received by a matching call to `recv()`.

Parameters:
- `dest` The process rank of the remote process to which the message will be sent.
- `tag` The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.

8.
```
template<typename T>
  status recv(int source, int tag, T & value) const;
```

Receive data from a remote process.

This routine blocks until it receives a message from the process `source` with the given `tag`. The type `T` of the `value` must be suitable for transmission over MPI, which includes serializable types, types that can be mapped to MPI data types (including most built-in C++ types), packed MPI archives, skeletons, and content associated with skeletons; see the documentation of `send` for a complete description.

Parameters:
- `source` The process that will be sending data. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.
- `tag` The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.
- `value` Will contain the value of the message after a successful receive. The type of this value must match the value transmitted by the sender, unless the sender transmitted a packed archive or skeleton: in these cases, the sender transmits a `packed_oarchive` or `packed_skeleton_oarchive` and the destination receives a `packed_iarchive` or `packed_skeleton_iarchive`, respectively.

Returns: Information about the received message.

9.
```
template<typename T>
  status recv(int source, int tag,
              const skeleton_proxy< T > & proxy) const;
```

Receive a skeleton from a remote process.

This routine blocks until it receives a message from the process `source` with the given `tag` containing a skeleton.

Parameters:
- `proxy` The `skeleton_proxy` containing a reference to the object that will be reshaped to match the received skeleton.
- `source` The process that will be sending data. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.
- `tag` The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.

Returns: Information about the received message.

10.
```
template<typename T>
  status recv(int source, int tag, skeleton_proxy< T > & proxy) const;
```

Receive a skeleton from a remote process.

This routine blocks until it receives a message from the process `source` with the given `tag` containing a skeleton.

Parameters:
- `proxy` The `skeleton_proxy` containing a reference to the object that will be reshaped to match the received skeleton.
- `source` The process that will be sending data. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.
- `tag` The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.

Returns: Information about the received message.

11.
```
template<typename T>
  status recv(int source, int tag, T * values, int n) const;
```

Receive an array of values from a remote process.

This routine blocks until it receives an array of values from the process `source` with the given `tag`. If the type `T` is

Parameters:
- `n` The number of values that can be stored into the `values` array. This shall not be smaller than the number of elements transmitted by the sender.
- `source` The process that will be sending data. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.
- `tag` The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.
- `values` Will contain the values in the message after a successful receive. The type of these elements must match the type of the elements transmitted by the sender.

Returns: Information about the received message.

Throws: std::range_error

12.
```
status recv(int source, int tag) const;
```

Receive a message from a remote process without any data.

This routine blocks until it receives a message from the process `source` with the given `tag`.

Parameters:
- `source` The process that will be sending the message. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.

	`tag`	The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.
Returns:	Information about the received message.	

13.
```
template<typename T>
  request isend(int dest, int tag, const T & value) const;
```

Send a message to a remote process without blocking.

The `isend` method is functionality identical to the `send` method and transmits data in the same way, except that `isend` will not block while waiting for the data to be transmitted. Instead, a request object will be immediately returned, allowing one to query the status of the communication or wait until it has completed.

Parameters:	`dest`	The rank of the remote process to which the data will be sent.
	`tag`	The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.
	`value`	The value that will be transmitted to the receiver. The type `T` of this value must meet the aforementioned criteria for transmission.
Returns:	a `request` object that describes this communication.	

14.
```
template<typename T>
  request isend(int dest, int tag,
                const skeleton_proxy< T > & proxy) const;
```

Send the skeleton of an object without blocking.

This routine is functionally identical to the `send` method for `skeleton_proxy` objects except that `isend` will not block while waiting for the data to be transmitted. Instead, a request object will be immediately returned, allowing one to query the status of the communication or wait until it has completed.

The semantics of this routine are equivalent to a non-blocking send of a `packed_skeleton_oarchive` storing the skeleton of the `object`.

Parameters:	`dest`	The rank of the remote process to which the skeleton will be sent.
	`proxy`	The `skeleton_proxy` containing a reference to the object whose skeleton will be transmitted.
	`tag`	The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.
Returns:	a `request` object that describes this communication.	

15.
```
template<typename T>
  request isend(int dest, int tag, const T * values, int n) const;
```

Send an array of values to another process without blocking.

This routine is functionally identical to the `send` method for arrays except that `isend` will not block while waiting for the data to be transmitted. Instead, a request object will be immediately returned, allowing one to query the status of the communication or wait until it has completed.

Parameters:	`dest`	The process rank of the remote process to which the data will be sent.
	`n`	The number of values stored in the array. The destination process must call receive with at least this many elements to correctly receive the message.
	`tag`	The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.
	`values`	The array of values that will be transmitted to the receiver. The type `T` of these values must be mapped to an MPI data type.
Returns:	a `request` object that describes this communication.	

16.
```
request isend(int dest, int tag) const;
```

Send a message to another process without any data without blocking.

This routine is functionally identical to the `send` method for sends with no data, except that `isend` will not block while waiting for the message to be transmitted. Instead, a request object will be immediately returned, allowing one to query the status of the communication or wait until it has completed.

Parameters:
- `dest` The process rank of the remote process to which the message will be sent.
- `tag` The tag that will be associated with this message. Tags may be any integer between zero and an implementation-defined upper limit. This limit is accessible via `environment::max_tag()`.

Returns: a `request` object that describes this communication.

17.
```
template<typename T>
  request irecv(int source, int tag, T & value) const;
```

Prepare to receive a message from a remote process.

The `irecv` method is functionally identical to the `recv` method and receive data in the same way, except that `irecv` will not block while waiting for data to be transmitted. Instead, it immediately returns a request object that allows one to query the status of the receive or wait until it has completed.

Parameters:
- `source` The process that will be sending data. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.
- `tag` The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.
- `value` Will contain the value of the message after a successful receive. The type of this value must match the value transmitted by the sender, unless the sender transmitted a packed archive or skeleton: in these cases, the sender transmits a `packed_oarchive` or `packed_skeleton_oarchive` and the destination receives a `packed_iarchive` or `packed_skeleton_iarchive`, respectively.

Returns: a `request` object that describes this communication.

18.
```
template<typename T>
  request irecv(int source, int tag, T * values, int n) const;
```

Initiate receipt of an array of values from a remote process.

This routine initiates a receive operation for an array of values transmitted by process `source` with the given `tag`.

Parameters:
- `n` The number of values that can be stored into the `values` array. This shall not be smaller than the number of elements transmitted by the sender.
- `source` The process that will be sending data. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.
- `tag` The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.
- `values` Will contain the values in the message after a successful receive. The type of these elements must match the type of the elements transmitted by the sender.

Returns: a `request` object that describes this communication.

19.
```
request irecv(int source, int tag) const;
```

Initiate receipt of a message from a remote process that carries no data.

This routine initiates a receive operation for a message from process `source` with the given `tag` that carries no data.

Parameters:
- `source` The process that will be sending the message. This will either be a process rank within the communicator or the constant `any_source`, indicating that we can receive the message from any process.
- `tag` The tag that matches a particular kind of message sent by the source process. This may be any tag value permitted by `send`. Alternatively, the argument may be the constant `any_tag`, indicating that this receive matches a message with any tag.

Returns: a `request` object that describes this communication.

20.
```
status probe(int source = any_source, int tag = any_tag) const;
```

Waits until a message is available to be received.

This operation waits until a message matching (`source`, `tag`) is available to be received. It then returns information about that message. The functionality is equivalent to `MPI_Probe`. To check if a message is available without blocking, use `iprobe`.

Parameters:
- `source` Determine if there is a message available from this rank. If `any_source`, then the message returned may come from any source.
- `tag` Determine if there is a message available with the given tag. If `any_tag`, then the message returned may have any tag.

Returns: Returns information about the first message that matches the given criteria.

21.
```
optional< status >
iprobe(int source = any_source, int tag = any_tag) const;
```

Determine if a message is available to be received.

This operation determines if a message matching (`source`, `tag`) is available to be received. If so, it returns information about that message; otherwise, it returns immediately with an empty optional. The functionality is equivalent to `MPI_Iprobe`. To wait until a message is available, use `wait`.

Parameters:
- `source` Determine if there is a message available from this rank. If `any_source`, then the message returned may come from any source.
- `tag` Determine if there is a message available with the given tag. If `any_tag`, then the message returned may have any tag.

Returns: If a matching message is available, returns information about that message. Otherwise, returns an empty `boost::optional`.

22.
```
void barrier() const;
```

Wait for all processes within a communicator to reach the barrier.

This routine is a collective operation that blocks each process until all processes have entered it, then releases all of the processes "simultaneously". It is equivalent to `MPI_Barrier`.

23.
```
operator bool() const;
```

Determine if this communicator is valid for communication.

Evaluates `true` in a boolean context if this communicator is valid for communication, i.e., does not represent MPI_COMM_NULL. Otherwise, evaluates `false`.

24.
```
operator MPI_Comm() const;
```

Access the MPI communicator associated with a Boost.MPI communicator.

This routine permits the implicit conversion from a Boost.MPI communicator to an MPI communicator.

Returns: The associated MPI communicator.

25.
```
communicator split(int color) const;
```

Split the communicator into multiple, disjoint communicators each of which is based on a particular color. This is a collective operation that returns a new communicator that is a subgroup of `this`. This routine is functionally equivalent to `MPI_Comm_split`.

Parameters: `color` The color of this process. All processes with the same `color` value will be placed into the same group.
Returns: A new communicator containing all of the processes in `this` that have the same `color`.

26.
```
communicator split(int color, int key) const;
```

Split the communicator into multiple, disjoint communicators each of which is based on a particular color. This is a collective operation that returns a new communicator that is a subgroup of `this`. This routine is functionally equivalent to `MPI_Comm_split`.

Parameters: `color` The color of this process. All processes with the same `color` value will be placed into the same group.
`key` A key value that will be used to determine the ordering of processes with the same color in the resulting communicator. If omitted, the rank of the processes in `this` will determine the ordering of processes in the resulting group.
Returns: A new communicator containing all of the processes in `this` that have the same `color`.

27.
```
optional< intercommunicator > as_intercommunicator() const;
```

Determine if the communicator is in fact an intercommunicator and, if so, return that intercommunicator.

Returns: an `optional` containing the intercommunicator, if this communicator is in fact an intercommunicator. Otherwise, returns an empty `optional`.

28.
```
optional< graph_communicator > as_graph_communicator() const;
```

Determine if the communicator has a graph topology and, if so, return that `graph_communicator`. Even though the communicators have different types, they refer to the same underlying communication space and can be used interchangeably for communication.

Returns: an `optional` containing the graph communicator, if this communicator does in fact have a graph topology. Otherwise, returns an empty `optional`.

29.
```
bool has_cartesian_topology() const;
```

Determines whether this communicator has a Cartesian topology.

30.
```
void abort(int errcode) const;
```

Abort all tasks in the group of this communicator.

Makes a "best attempt" to abort all of the tasks in the group of this communicator. Depending on the underlying MPI implementation, this may either abort the entire program (and possibly return `errcode` to the environment) or only abort some processes, allowing the others to continue. Consult the documentation for your MPI implementation. This is equivalent to a call to `MPI_Abort`

Parameters: `errcode` The error code to return from aborted processes.
Returns: Will not return.

Type comm_create_kind

boost::mpi::comm_create_kind — Enumeration used to describe how to adopt a C `MPI_Comm` into a Boost.MPI communicator.

Synopsis

```
// In header: <boost/mpi/communicator.hpp>


enum comm_create_kind { comm_duplicate, comm_take_ownership,
                        comm_attach };
```

Description

The values for this enumeration determine how a Boost.MPI communicator will behave when constructed with an MPI communicator. The options are:

- `comm_duplicate`: Duplicate the MPI_Comm communicator to create a new communicator (e.g., with MPI_Comm_dup). This new MPI_Comm communicator will be automatically freed when the Boost.MPI communicator (and all copies of it) is destroyed.
- `comm_take_ownership`: Take ownership of the communicator. It will be freed automatically when all of the Boost.MPI communicators go out of scope. This option must not be used with MPI_COMM_WORLD.
- `comm_attach`: The Boost.MPI communicator will reference the existing MPI communicator but will not free it when the Boost.MPI communicator goes out of scope. This option should only be used when the communicator is managed by the user or MPI library (e.g., MPI_COMM_WORLD).

Global any_source

boost::mpi::any_source — A constant representing "any process.".

Synopsis

```
// In header: <boost/mpi/communicator.hpp>

const int any_source;
```

Description

This constant may be used for the `source` parameter of `receive` operations to indicate that a message may be received from any source.

Global any_tag

boost::mpi::any_tag — A constant representing "any tag.".

Synopsis

```
// In header: <boost/mpi/communicator.hpp>

const int any_tag;
```

Description

This constant may be used for the `tag` parameter of `receive` operations to indicate that a `send` with any tag will be matched by the receive.

Function operator==

boost::mpi::operator== — Determines whether two communicators are identical.

Synopsis

```
// In header: <boost/mpi/communicator.hpp>


BOOST_MPI_DECL bool
operator==(const communicator & comm1, const communicator & comm2);
```

Description

Equivalent to calling `MPI_Comm_compare` and checking whether the result is `MPI_IDENT`.

Returns: True when the two communicators refer to the same underlying MPI communicator.

Function operator!=

boost::mpi::operator!= — Determines whether two communicators are different.

Synopsis

```
// In header: <boost/mpi/communicator.hpp>


bool operator!=(const communicator & comm1,
                const communicator & comm2);
```

Description

Returns: !(comm1 == comm2)

Header <boost/mpi/config.hpp>

This header provides MPI configuration details that expose the capabilities of the underlying MPI implementation, and provides auto-linking support on Windows.

```


MPICH_IGNORE_CXX_SEEK
BOOST_MPI_HAS_MEMORY_ALLOCATION
BOOST_MPI_HAS_NOARG_INITIALIZATION
BOOST_MPI_CALLING_CONVENTION
```

Macro MPICH_IGNORE_CXX_SEEK

MPICH_IGNORE_CXX_SEEK

Synopsis

```
// In header: <boost/mpi/config.hpp>

MPICH_IGNORE_CXX_SEEK
```

Macro BOOST_MPI_HAS_MEMORY_ALLOCATION

BOOST_MPI_HAS_MEMORY_ALLOCATION — Define this macro to avoid expensice MPI_Pack/Unpack calls on homogeneous machines.

Synopsis

```
// In header: <boost/mpi/config.hpp>

BOOST_MPI_HAS_MEMORY_ALLOCATION
```

Description

Determine if the MPI implementation has support for memory allocation. This macro will be defined when the underlying MPI implementation has support for the MPI-2 memory allocation routines `MPI_Alloc_mem` and `MPI_Free_mem`. When defined, the `allocator` class template will provide Standard Library-compliant access to these memory-allocation routines.

Macro BOOST_MPI_HAS_NOARG_INITIALIZATION

BOOST_MPI_HAS_NOARG_INITIALIZATION — Determine if the MPI implementation has supports initialization without command-line arguments.

Synopsis

```
// In header: <boost/mpi/config.hpp>

BOOST_MPI_HAS_NOARG_INITIALIZATION
```

Description

This macro will be defined when the underlying implementation supports initialization of MPI without passing along command-line arguments, e.g., `MPI_Init(NULL, NULL)`. When defined, the `environment` class will provide a default constructor. This macro is always defined for MPI-2 implementations.

Macro BOOST_MPI_CALLING_CONVENTION

BOOST_MPI_CALLING_CONVENTION — Specifies the calling convention that will be used for callbacks from the underlying C MPI.

Synopsis

```
// In header: <boost/mpi/config.hpp>

BOOST_MPI_CALLING_CONVENTION
```

Description

This is a Windows-specific macro, which will be used internally to state the calling convention of any function that is to be used as a callback from MPI. For example, the internally-defined functions that are used in a call to `MPI_Op_create`. This macro is likely only to be useful to users that wish to bypass Boost.MPI, registering their own callbacks in certain cases, e.g., through `MPI_Op_create`.

Header <boost/mpi/datatype.hpp>

This header provides the mapping from C++ types to MPI data types.

```


BOOST_IS_MPI_DATATYPE(T)
```

```
namespace boost {
  namespace mpi {
    template<typename T> struct is_mpi_integer_datatype;
    template<typename T> struct is_mpi_floating_point_datatype;
    template<typename T> struct is_mpi_logical_datatype;
    template<typename T> struct is_mpi_complex_datatype;
    template<typename T> struct is_mpi_byte_datatype;
    template<typename T> struct is_mpi_builtin_datatype;
    template<typename T> struct is_mpi_datatype;
    template<typename T> MPI_Datatype get_mpi_datatype(const T &);
  }
}
```

Struct template is_mpi_integer_datatype

boost::mpi::is_mpi_integer_datatype — Type trait that determines if there exists a built-in integer MPI data type for a given C++ type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T>
struct is_mpi_integer_datatype {
};
```

Description

This ytpe trait determines when there is a direct mapping from a C++ type to an MPI data type that is classified as an integer data type. See `is_mpi_builtin_datatype` for general information about built-in MPI data types.

Struct template is_mpi_floating_point_datatype

boost::mpi::is_mpi_floating_point_datatype — Type trait that determines if there exists a built-in floating point MPI data type for a given C++ type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T>
struct is_mpi_floating_point_datatype {
};
```

Description

This ytpe trait determines when there is a direct mapping from a C++ type to an MPI data type that is classified as a floating point data type. See `is_mpi_builtin_datatype` for general information about built-in MPI data types.

Struct template is_mpi_logical_datatype

boost::mpi::is_mpi_logical_datatype — Type trait that determines if there exists a built-in logical MPI data type for a given C++ type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T>
struct is_mpi_logical_datatype {
};
```

Description

This ytpe trait determines when there is a direct mapping from a C++ type to an MPI data type that is classified as an logical data type. See `is_mpi_builtin_datatype` for general information about built-in MPI data types.

Struct template is_mpi_complex_datatype

boost::mpi::is_mpi_complex_datatype — Type trait that determines if there exists a built-in complex MPI data type for a given C++ type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T>
struct is_mpi_complex_datatype {
};
```

Description

This ytpe trait determines when there is a direct mapping from a C++ type to an MPI data type that is classified as an complex data type. See `is_mpi_builtin_datatype` for general information about built-in MPI data types.

Struct template is_mpi_byte_datatype

boost::mpi::is_mpi_byte_datatype — Type trait that determines if there exists a built-in byte MPI data type for a given C++ type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T>
struct is_mpi_byte_datatype {
};
```

Description

This ytpe trait determines when there is a direct mapping from a C++ type to an MPI data type that is classified as an byte data type. See `is_mpi_builtin_datatype` for general information about built-in MPI data types.

Struct template is_mpi_builtin_datatype

boost::mpi::is_mpi_builtin_datatype — Type trait that determines if there exists a built-in MPI data type for a given C++ type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T>
struct is_mpi_builtin_datatype {
};
```

Description

This type trait determines when there is a direct mapping from a C++ type to an MPI type. For instance, the C++ `int` type maps directly to the MPI type `MPI_INT`. When there is a direct mapping from the type `T` to an MPI type, `is_mpi_builtin_datatype` will derive from `mpl::true_` and the MPI data type will be accessible via `get_mpi_datatype`.

In general, users should not need to specialize this trait. However, if you have an additional C++ type that can map directly to only of MPI's built-in types, specialize either this trait or one of the traits corresponding to categories of MPI data types (`is_mpi_integer_datatype`, `is_mpi_floating_point_datatype`, `is_mpi_logical_datatype`, `is_mpi_complex_datatype`, or `is_mpi_builtin_datatype`). `is_mpi_builtin_datatype` derives `mpl::true_` if any of the traits corresponding to MPI data type categories derived `mpl::true_`.

Struct template is_mpi_datatype

boost::mpi::is_mpi_datatype — Type trait that determines if a C++ type can be mapped to an MPI data type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T>
struct is_mpi_datatype :
  public boost::mpi::is_mpi_builtin_datatype< T >
{
};
```

Description

This type trait determines if it is possible to build an MPI data type that represents a C++ data type. When this is the case, `is_mpi_datatype` derives `mpl::true_` and the MPI data type will be accessible via `get_mpi_datatype`.

For any C++ type that maps to a built-in MPI data type (see `is_mpi_builtin_datatype`), `is_mpi_data_type` is trivially true. However, any POD ("Plain Old Data") type containing types that themselves can be represented by MPI data types can itself be represented as an MPI data type. For instance, a `point3d` class containing three `double` values can be represented as an MPI data type. To do so, first make the data type Serializable (using the Boost.Serialization library); then, specialize the `is_mpi_datatype` trait for the point type so that it will derive `mpl::true_`:

```
    namespace boost { namespace mpi {
      template<> struct is_mpi_datatype<point>
        : public mpl::true_ { };
    } }
```

Function template get_mpi_datatype

boost::mpi::get_mpi_datatype — Returns an MPI data type for a C++ type.

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

template<typename T> MPI_Datatype get_mpi_datatype(const T & x);
```

Description

The function creates an MPI data type for the given object `x`. The first time it is called for a class `T`, the MPI data type is created and cached. Subsequent calls for objects of the same type `T` return the cached MPI data type. The type `T` must allow creation of an MPI data type. That is, it must be Serializable and `is_mpi_datatype<T>` must derive `mpl::true_`.

For fundamental MPI types, a copy of the MPI data type of the MPI library is returned.

Note that since the data types are cached, the caller should never call `MPI_Type_free()` for the MPI data type returned by this call.

Parameters: `x` for an optimized call, a constructed object of the type should be passed; otherwise, an object will be default-constructed.

Returns: The MPI data type corresponding to type `T`.

Macro BOOST_IS_MPI_DATATYPE

BOOST_IS_MPI_DATATYPE

Synopsis

```
// In header: <boost/mpi/datatype.hpp>

BOOST_IS_MPI_DATATYPE(T)
```

Header <boost/mpi/datatype_fwd.hpp>

This header provides forward declarations for the contents of the header `datatype.hpp`. It is expected to be used primarily by user-defined C++ classes that need to specialize `is_mpi_datatype`.

```
namespace boost {
  namespace mpi {
    struct packed;
    template<typename T> MPI_Datatype get_mpi_datatype();
  }
}
```

Struct packed

boost::mpi::packed — a dummy data type giving MPI_PACKED as its MPI_Datatype

Synopsis

```
// In header: <boost/mpi/datatype_fwd.hpp>


struct packed {
};
```

Header <boost/mpi/environment.hpp>

This header provides the `environment` class, which provides routines to initialize, finalization, and query the status of the Boost MPI environment.

```
namespace boost {
  namespace mpi {
    class environment;
  }
}
```

Class environment

boost::mpi::environment — Initialize, finalize, and query the MPI environment.

Synopsis

```
// In header: <boost/mpi/environment.hpp>


class environment {
public:
  // construct/copy/destruct
  environment(bool = true);
  environment(int &, char **&, bool = true);
  ~environment();

  // public static functions
  static void abort(int);
  static bool initialized();
  static bool finalized();
  static int max_tag();
  static int collectives_tag();
  static optional< int > host_rank();
  static optional< int > io_rank();
  static std::string processor_name();
};
```

Description

The `environment` class is used to initialize, finalize, and query the MPI environment. It will typically be used in the `main()` function of a program, which will create a single instance of `environment` initialized with the arguments passed to the program:

```
  int main(int argc, char* argv[])
  {
    mpi::environment env(argc, argv);
  }
```

The instance of `environment` will initialize MPI (by calling `MPI_Init`) in its constructor and finalize MPI (by calling `MPI_Finalize` for normal termination or `MPI_Abort` for an uncaught exception) in its destructor.

The use of `environment` is not mandatory. Users may choose to invoke `MPI_Init` and `MPI_Finalize` manually. In this case, no `environment` object is needed. If one is created, however, it will do nothing on either construction or destruction.

`environment` public construct/copy/destruct

1. ```
   environment(bool abort_on_exception = true);
   ```

   Initialize the MPI environment.

   If the MPI environment has not already been initialized, initializes MPI with a call to `MPI_Init`. Since this constructor does not take command-line arguments (`argc` and `argv`), it is only available when the underlying MPI implementation supports calling `MPI_Init` with `NULL` arguments, indicated by the macro `BOOST_MPI_HAS_NOARG_INITIALIZATION`.

   Parameters: `abort_on_exception` — When true, this object will abort the program if it is destructed due to an uncaught exception.
   ```

2.
```
environment(int & argc, char **& argv,
            bool abort_on_exception = true);
```

Initialize the MPI environment.

If the MPI environment has not already been initialized, initializes MPI with a call to `MPI_Init`.

Parameters:	`abort_on_exception`	When true, this object will abort the program if it is destructed due to an uncaught exception.
	`argc`	The number of arguments provided in `argv`, as passed into the program's `main` function.
	`argv`	The array of argument strings passed to the program via `main`.

3.
```
~environment();
```

Shuts down the MPI environment.

If this `environment` object was used to initialize the MPI environment, and the MPI environment has not already been shut down (finalized), this destructor will shut down the MPI environment. Under normal circumstances, this only involves invoking `MPI_Finalize`. However, if destruction is the result of an uncaught exception and the `abort_on_exception` parameter of the constructor had the value `true`, this destructor will invoke `MPI_Abort` with `MPI_COMM_WORLD` to abort the entire MPI program with a result code of -1.

`environment` public static functions

1.
```
static void abort(int errcode);
```

Abort all MPI processes.

Aborts all MPI processes and returns to the environment. The precise behavior will be defined by the underlying MPI implementation. This is equivalent to a call to `MPI_Abort` with `MPI_COMM_WORLD`.

Parameters:	`errcode`	The error code to return to the environment.
Returns:	Will not return.	

2.
```
static bool initialized();
```

Determine if the MPI environment has already been initialized.

This routine is equivalent to a call to `MPI_Initialized`.

Returns: `true` if the MPI environment has been initialized.

3.
```
static bool finalized();
```

Determine if the MPI environment has already been finalized.

The routine is equivalent to a call to `MPI_Finalized`.

Returns: `true` if the MPI environment has been finalized.

4.
```
static int max_tag();
```

Retrieves the maximum tag value.

Returns the maximum value that may be used for the `tag` parameter of send/receive operations. This value will be somewhat smaller than the value of `MPI_TAG_UB`, because the Boost.MPI implementation reserves some tags for collective operations.

Returns: the maximum tag value.

5.
```
static int collectives_tag();
```

The tag value used for collective operations.

Returns the reserved tag value used by the Boost.MPI implementation for collective operations. Although users are not permitted to use this tag to send or receive messages, it may be useful when monitoring communication patterns.

Returns: the tag value used for collective operations.

6.
```
static optional< int > host_rank();
```

Retrieves the rank of the host process, if one exists.

If there is a host process, this routine returns the rank of that process. Otherwise, it returns an empty `optional<int>`. MPI does not define the meaning of a "host" process: consult the documentation for the MPI implementation. This routine examines the `MPI_HOST` attribute of `MPI_COMM_WORLD`.

Returns: The rank of the host process, if one exists.

7.
```
static optional< int > io_rank();
```

Retrieves the rank of a process that can perform input/output.

This routine returns the rank of a process that can perform input/output via the standard C and C++ I/O facilities. If every process can perform I/O using the standard facilities, this routine will return `any_source`; if no process can perform I/O, this routine will return no value (an empty `optional`). This routine examines the `MPI_IO` attribute of `MPI_COMM_WORLD`.

Returns: the rank of the process that can perform I/O, `any_source` if every process can perform I/O, or no value if no process can perform I/O.

8.
```
static std::string processor_name();
```

Retrieve the name of this processor.

This routine returns the name of this processor. The actual form of the name is unspecified, but may be documented by the underlying MPI implementation. This routine is implemented as a call to `MPI_Get_processor_name`.

Returns: the name of this processor.

Header <boost/mpi/exception.hpp>

This header provides exception classes that report MPI errors to the user and macros that translate MPI error codes into Boost.MPI exceptions.

```
BOOST_MPI_CHECK_RESULT(MPIFunc, Args)
```

```
namespace boost {
  namespace mpi {
    class exception;
  }
}
```

Class exception

boost::mpi::exception — Catch-all exception class for MPI errors.

Synopsis

```
// In header: <boost/mpi/exception.hpp>


class exception {
public:
  // construct/copy/destruct
  exception(const char *, int);
  ~exception();

  // public member functions
  const char * what() const;
  const char * routine() const;
  int result_code() const;
  int error_class() const;
};
```

Description

Instances of this class will be thrown when an MPI error occurs. MPI failures that trigger these exceptions may or may not be recoverable, depending on the underlying MPI implementation. Consult the documentation for your MPI implementation to determine the effect of MPI errors.

`exception` public construct/copy/destruct

1. ```
   exception(const char * routine, int result_code);
   ```

   Build a new `exception` exception.

   Parameters:

`result_code`	The result code returned from the MPI routine that aborted with an error.
`routine`	The MPI routine in which the error occurred. This should be a pointer to a string constant: it will not be copied.

2. ```
   ~exception();
   ```

`exception` public member functions

1. ```
   const char * what() const;
   ```

   A description of the error that occurred.

2. ```
   const char * routine() const;
   ```

   Retrieve the name of the MPI routine that reported the error.

3. ```
   int result_code() const;
   ```

   Retrieve the result code returned from the MPI routine that reported the error.
   ```

4.
```
int error_class() const;
```

Returns the MPI error class associated with the error that triggered this exception.

Macro BOOST_MPI_CHECK_RESULT

BOOST_MPI_CHECK_RESULT

Synopsis

```
// In header: <boost/mpi/exception.hpp>

BOOST_MPI_CHECK_RESULT(MPIFunc, Args)
```

Description

Call the MPI routine MPIFunc with arguments Args (surrounded by parentheses). If the result is not MPI_SUCCESS, use boost::throw_exception to throw an exception or abort, depending on BOOST_NO_EXCEPTIONS.

Header <boost/mpi/graph_communicator.hpp>

This header defines facilities to support MPI communicators with graph topologies, using the graph interface defined by the Boost Graph Library. One can construct a communicator whose topology is described by any graph meeting the requirements of the Boost Graph Library's graph concepts. Likewise, any communicator that has a graph topology can be viewed as a graph by the Boost Graph Library, permitting one to use the BGL's graph algorithms on the process topology.

```
namespace boost {
  template<> struct graph_traits<mpi::graph_communicator>;
  namespace mpi {
    class graph_communicator;

    // Returns the source vertex from an edge in the graph topology of a communicator.
    int source(const std::pair< int, int > & edge,
               const graph_communicator &);

    // Returns the target vertex from an edge in the graph topology of a communicator.
    int target(const std::pair< int, int > & edge,
               const graph_communicator &);

    // Returns an iterator range containing all of the edges outgoing from the given vertex in ↵
a graph topology of a communicator.
    unspecified out_edges(int vertex,
                          const graph_communicator & comm);

    // Returns the out-degree of a vertex in the graph topology of a communicator.
    int out_degree(int vertex, const graph_communicator & comm);

    // Returns an iterator range containing all of the neighbors of the given vertex in the com↵
municator's graph topology.
    unspecified adjacent_vertices(int vertex,
                                  const graph_communicator & comm);

    // Returns an iterator range that contains all of the vertices with the communicator's graph ↵
topology, i.e., all of the process ranks in the communicator.
    std::pair< counting_iterator< int >, counting_iterator< int > >
    vertices(const graph_communicator & comm);

    // Returns the number of vertices within the graph topology of the communicator, i.e., the ↵
number of processes in the communicator.
    int num_vertices(const graph_communicator & comm);

    // Returns an iterator range that contains all of the edges with the communicator's graph ↵
topology.
    unspecified edges(const graph_communicator & comm);

    // Returns the number of edges in the communicator's graph topology.
    int num_edges(const graph_communicator & comm);
    identity_property_map
    get(vertex_index_t, const graph_communicator &);
    int get(vertex_index_t, const graph_communicator &, int);
  }
}
```

Class graph_communicator

boost::mpi::graph_communicator — An MPI communicator with a graph topology.

Synopsis

```
// In header: <boost/mpi/graph_communicator.hpp>


class graph_communicator : public boost::mpi::communicator {
public:
  // construct/copy/destruct
  graph_communicator(const MPI_Comm &, comm_create_kind);
  template<typename Graph>
    graph_communicator(const communicator &, const Graph &,
                       bool = false);
  template<typename Graph, typename RankMap>
    graph_communicator(const communicator &, const Graph &,
                       RankMap, bool = false);
};
```

Description

A `graph_communicator` is a communicator whose topology is expressed as a graph. Graph communicators have the same functionality as (intra)communicators, but also allow one to query the relationships among processes. Those relationships are expressed via a graph, using the interface defined by the Boost Graph Library. The `graph_communicator` class meets the requirements of the BGL Graph, Incidence Graph, Adjacency Graph, Vertex List Graph, and Edge List Graph concepts.

`graph_communicator` public construct/copy/destruct

1.
```
graph_communicator(const MPI_Comm & comm, comm_create_kind kind);
```

Build a new Boost.MPI graph communicator based on the MPI communicator `comm` with graph topology.

`comm` may be any valid MPI communicator. If `comm` is MPI_COMM_NULL, an empty communicator (that cannot be used for communication) is created and the `kind` parameter is ignored. Otherwise, the `kind` parameter determines how the Boost.MPI communicator will be related to `comm`:

- If `kind` is `comm_duplicate`, duplicate `comm` to create a new communicator. This new communicator will be freed when the Boost.MPI communicator (and all copies of it) is destroyed. This option is only permitted if the underlying MPI implementation supports MPI 2.0; duplication of intercommunicators is not available in MPI 1.x.
- If `kind` is `comm_take_ownership`, take ownership of `comm`. It will be freed automatically when all of the Boost.MPI communicators go out of scope.
- If `kind` is `comm_attach`, this Boost.MPI communicator will reference the existing MPI communicator `comm` but will not free `comm` when the Boost.MPI communicator goes out of scope. This option should only be used when the communicator is managed by the user.

2.
```
template<typename Graph>
  graph_communicator(const communicator & comm,
                     const Graph & graph, bool reorder = false);
```

Create a new communicator whose topology is described by the given graph. The indices of the vertices in the graph will be assumed to be the ranks of the processes within the communicator. There may be fewer vertices in the graph than there are processes in the communicator; in this case, the resulting communicator will be a NULL communicator.

Parameters: `comm` The communicator that the new, graph communicator will be based on.

`graph` Any type that meets the requirements of the Incidence Graph and Vertex List Graph concepts from the Boost Graph Library. This structure of this graph will become the topology of the communicator that is returned.

`reorder` Whether MPI is permitted to re-order the process ranks within the returned communicator, to better optimize communication. If false, the ranks of each process in the returned process will match precisely the rank of that process within the original communicator.

3.
```
template<typename Graph, typename RankMap>
  graph_communicator(const communicator & comm,
                     const Graph & graph, RankMap rank,
                     bool reorder = false);
```

Create a new communicator whose topology is described by the given graph. The rank map (`rank`) gives the mapping from vertices in the graph to ranks within the communicator. There may be fewer vertices in the graph than there are processes in the communicator; in this case, the resulting communicator will be a NULL communicator.

Parameters:

`comm` The communicator that the new, graph communicator will be based on. The ranks in `rank` refer to the processes in this communicator.

`graph` Any type that meets the requirements of the Incidence Graph and Vertex List Graph concepts from the Boost Graph Library. This structure of this graph will become the topology of the communicator that is returned.

`rank` This map translates vertices in the `graph` into ranks within the current communicator. It must be a Readable Property Map (see the Boost Property Map library) whose key type is the vertex type of the `graph` and whose value type is `int`.

`reorder` Whether MPI is permitted to re-order the process ranks within the returned communicator, to better optimize communication. If false, the ranks of each process in the returned process will match precisely the rank of that process within the original communicator.

Function get

boost::mpi::get — Returns a property map that maps from vertices in a communicator's graph topology to their index values.

Synopsis

```
// In header: <boost/mpi/graph_communicator.hpp>


identity_property_map
get(vertex_index_t, const graph_communicator &);
```

Description

Since the vertices are ranks in the communicator, the returned property map is the identity property map.

Function get

boost::mpi::get — Returns the index of a vertex in the communicator's graph topology.

Synopsis

```
// In header: <boost/mpi/graph_communicator.hpp>

int get(vertex_index_t, const graph_communicator &, int vertex);
```

Description

Since the vertices are ranks in the communicator, this is the identity function.

Struct graph_traits<mpi::graph_communicator>

boost::graph_traits<mpi::graph_communicator> — Traits structure that allows a communicator with graph topology to be view as a graph by the Boost Graph Library.

Synopsis

```
// In header: <boost/mpi/graph_communicator.hpp>


struct graph_traits<mpi::graph_communicator> {
  // types
  typedef int                          vertex_descriptor;
  typedef std::pair< int, int >        edge_descriptor;
  typedef directed_tag                 directed_category;
  typedef disallow_parallel_edge_tag edge_parallel_category;
  typedef unspecified                  out_edge_iterator;
  typedef int                          degree_size_type;
  typedef unspecified                  adjacency_iterator;
  typedef counting_iterator< int >     vertex_iterator;
  typedef int                          vertices_size_type;
  typedef unspecified                  edge_iterator;
  typedef int                          edges_size_type;

  // public static functions
  static vertex_descriptor null_vertex();
};
```

Description

The specialization of `graph_traits` for an MPI communicator allows a communicator with graph topology to be viewed as a graph. An MPI communicator with graph topology meets the requirements of the Graph, Incidence Graph, Adjacency Graph, Vertex List Graph, and Edge List Graph concepts from the Boost Graph Library.

`graph_traits` public static functions

1. ```
   static vertex_descriptor null_vertex();
   ```

   Returns a vertex descriptor that can never refer to any valid vertex.

Header <boost/mpi/group.hpp>

This header defines the `group` class, which allows one to manipulate and query groups of processes.

```
namespace boost {
  namespace mpi {
    class group;
    BOOST_MPI_DECL bool operator==(const group &, const group &);
    bool operator!=(const group &, const group &);
    BOOST_MPI_DECL group operator|(const group &, const group &);
    BOOST_MPI_DECL group operator&(const group &, const group &);
    BOOST_MPI_DECL group operator-(const group &, const group &);
  }
}
```
```

Class group

boost::mpi::group — A `group` is a representation of a subset of the processes within a `communicator`.

Synopsis

```
// In header: <boost/mpi/group.hpp>


class group {
public:
  // construct/copy/destruct
  group();
  group(const MPI_Group &, bool);

  // public member functions
  optional< int > rank() const;
  int size() const;
  template<typename InputIterator, typename OutputIterator>
    OutputIterator
    translate_ranks(InputIterator, InputIterator, const group &,
                    OutputIterator);
  operator bool() const;
  operator MPI_Group() const;
  template<typename InputIterator>
    group include(InputIterator, InputIterator);
  template<typename InputIterator>
    group exclude(InputIterator, InputIterator);
};
```

Description

The `group` class allows one to create arbitrary subsets of the processes within a communicator. One can compute the union, intersection, or difference of two groups, or create new groups by specifically including or excluding certain processes. Given a group, one can create a new communicator containing only the processes in that group.

`group` public construct/copy/destruct

1. ```
   group();
   ```

   Constructs an empty group.

2. ```
   group(const MPI_Group & in_group, bool adopt);
   ```

   Constructs a group from an `MPI_Group`.

   This routine allows one to construct a Boost.MPI `group` from a C `MPI_Group`. The `group` object can (optionally) adopt the `MPI_Group`, after which point the `group` object becomes responsible for freeing the `MPI_Group` when the last copy of `group` disappears.

   Parameters:

`adopt`	Whether the `group` should adopt the `MPI_Group`. When true, the `group` object (or one of its copies) will free the group (via `MPI_Comm_free`) when the last copy is destroyed. Otherwise, the user is responsible for calling `MPI_Group_free`.
`in_group`	The `MPI_Group` used to construct this `group`.

`group` public member functions

1.
```
optional< int > rank() const;
```

Determine the rank of the calling process in the group.

This routine is equivalent to `MPI_Group_rank`.

Returns: The rank of the calling process in the group, which will be a value in [0, size()). If the calling process is not in the group, returns an empty value.

2.
```
int size() const;
```

Determine the number of processes in the group.

This routine is equivalent to `MPI_Group_size`.

Returns: The number of processes in the group.

3.
```
template<typename InputIterator, typename OutputIterator>
  OutputIterator
  translate_ranks(InputIterator first, InputIterator last,
                  const group & to_group, OutputIterator out);
```

Translates the ranks from one group into the ranks of the same processes in another group.

This routine translates each of the integer rank values in the iterator range [first, last) from the current group into rank values of the corresponding processes in `to_group`. The corresponding rank values are written via the output iterator `out`. When there is no correspondence between a rank in the current group and a rank in `to_group`, the value `MPI_UNDEFINED` is written to the output iterator.

Parameters:

`first`	Beginning of the iterator range of ranks in the current group.
`last`	Past the end of the iterator range of ranks in the current group.
`out`	The output iterator to which the translated ranks will be written.
`to_group`	The group that we are translating ranks to.

Returns: the output iterator, which points one step past the last rank written.

4.
```
operator bool() const;
```

Determines whether the group is non-empty.

Returns: True if the group is not empty, false if it is empty.

5.
```
operator MPI_Group() const;
```

Retrieves the underlying `MPI_Group` associated with this group.

Returns: The `MPI_Group` handle manipulated by this object. If this object represents the empty group, returns `MPI_GROUP_EMPTY`.

6.
```
template<typename InputIterator>
  group include(InputIterator first, InputIterator last);
```

Creates a new group including a subset of the processes in the current group.

This routine creates a new `group` which includes only those processes in the current group that are listed in the integer iterator range [first, last). Equivalent to `MPI_Group_incl`.

`first` The beginning of the iterator range of ranks to include.

`last` Past the end of the iterator range of ranks to include.

Returns: A new group containing those processes with ranks [first, last) in the current group.

7.
```
template<typename InputIterator>
  group exclude(InputIterator first, InputIterator last);
```

Creates a new group from all of the processes in the current group, exluding a specific subset of the processes.

This routine creates a new `group` which includes all of the processes in the current group except those whose ranks are listed in the integer iterator range [first, last). Equivalent to `MPI_Group_excl`.

`first` The beginning of the iterator range of ranks to exclude.

`last` Past the end of the iterator range of ranks to exclude.

Returns: A new group containing all of the processes in the current group except those processes with ranks [first, last) in the current group.

Function operator==

boost::mpi::operator== — Determines whether two process groups are identical.

Synopsis

```
// In header: <boost/mpi/group.hpp>

BOOST_MPI_DECL bool operator==(const group & g1, const group & g2);
```

Description

Equivalent to calling `MPI_Group_compare` and checking whether the result is `MPI_IDENT`.

Returns: True when the two process groups contain the same processes in the same order.

Function operator!=

boost::mpi::operator!= — Determines whether two process groups are not identical.

Synopsis

```
// In header: <boost/mpi/group.hpp>

bool operator!=(const group & g1, const group & g2);
```

Description

Equivalent to calling `MPI_Group_compare` and checking whether the result is not `MPI_IDENT`.

Returns: False when the two process groups contain the same processes in the same order.

Function operator|

boost::mpi::operator| — Computes the union of two process groups.

Synopsis

```
// In header: <boost/mpi/group.hpp>


BOOST_MPI_DECL group operator|(const group & g1, const group & g2);
```

Description

This routine returns a new `group` that contains all processes that are either in group `g1` or in group `g2` (or both). The processes that are in `g1` will be first in the resulting group, followed by the processes from `g2` (but not also in `g1`). Equivalent to `MPI_Group_union`.

Function operator&

boost::mpi::operator& — Computes the intersection of two process groups.

Synopsis

```
// In header: <boost/mpi/group.hpp>


BOOST_MPI_DECL group operator&(const group & g1, const group & g2);
```

Description

This routine returns a new `group` that contains all processes that are in group `g1` and in group `g2`, ordered in the same way as `g1`. Equivalent to `MPI_Group_intersection`.

Function operator-

boost::mpi::operator- — Computes the difference between two process groups.

Synopsis

```
// In header: <boost/mpi/group.hpp>


BOOST_MPI_DECL group operator-(const group & g1, const group & g2);
```

Description

This routine returns a new `group` that contains all processes that are in group `g1` but not in group `g2`, ordered in the same way as `g1`. Equivalent to `MPI_Group_difference`.

Header <boost/mpi/intercommunicator.hpp>

This header defines the `intercommunicator` class, which permits communication between different process groups.

```
namespace boost {
  namespace mpi {
    class intercommunicator;
  }
}
```

Class intercommunicator

boost::mpi::intercommunicator — Communication facilities among processes in different groups.

Synopsis

```
// In header: <boost/mpi/intercommunicator.hpp>


class intercommunicator : public boost::mpi::communicator {
public:
  // construct/copy/destruct
  intercommunicator(const MPI_Comm &, comm_create_kind);
  intercommunicator(const communicator &, int,
                    const communicator &, int);

  // public member functions
  int local_size() const;
  boost::mpi::group local_group() const;
  int local_rank() const;
  int remote_size() const;
  boost::mpi::group remote_group() const;
  communicator merge(bool) const;
};
```

Description

The `intercommunicator` class provides communication facilities among processes from different groups. An intercommunicator is always associated with two process groups: one "local" process group, containing the process that initiates an MPI operation (e.g., the sender in a `send` operation), and one "remote" process group, containing the process that is the target of the MPI operation.

While intercommunicators have essentially the same point-to-point operations as intracommunicators (the latter communicate only within a single process group), all communication with intercommunicators occurs between the processes in the local group and the processes in the remote group; communication within a group must use a different (intra-)communicator.

`intercommunicator` public construct/copy/destruct

1. ```
   intercommunicator(const MPI_Comm & comm, comm_create_kind kind);
   ```

   Build a new Boost.MPI intercommunicator based on the MPI intercommunicator `comm`.

   `comm` may be any valid MPI intercommunicator. If `comm` is MPI_COMM_NULL, an empty communicator (that cannot be used for communication) is created and the `kind` parameter is ignored. Otherwise, the `kind` parameter determines how the Boost.MPI communicator will be related to `comm`:

   - If `kind` is `comm_duplicate`, duplicate `comm` to create a new communicator. This new communicator will be freed when the Boost.MPI communicator (and all copies of it) is destroyed. This option is only permitted if the underlying MPI implementation supports MPI 2.0; duplication of intercommunicators is not available in MPI 1.x.
   - If `kind` is `comm_take_ownership`, take ownership of `comm`. It will be freed automatically when all of the Boost.MPI communicators go out of scope.
   - If `kind` is `comm_attach`, this Boost.MPI communicator will reference the existing MPI communicator `comm` but will not free `comm` when the Boost.MPI communicator goes out of scope. This option should only be used when the communicator is managed by the user.

2. ```
   intercommunicator(const communicator & local, int local_leader,
                     const communicator & peer, int remote_leader);
   ```

Constructs a new intercommunicator whose local group is `local` and whose remote group is `peer`. The intercommunicator can then be used to communicate between processes in the two groups. This constructor is equivalent to a call to `MPI_Intercomm_create`.

Parameters:		
	`local`	The intracommunicator containing all of the processes that will go into the local group.
	`local_leader`	The rank within the `local` intracommunicator that will serve as its leader.
	`peer`	The intracommunicator containing all of the processes that will go into the remote group.
	`remote_leader`	The rank within the `peer` group that will serve as its leader.

`intercommunicator` public member functions

1.
```
int local_size() const;
```

Returns the size of the local group, i.e., the number of local processes that are part of the group.

2.
```
boost::mpi::group local_group() const;
```

Returns the local group, containing all of the local processes in this intercommunicator.

3.
```
int local_rank() const;
```

Returns the rank of this process within the local group.

4.
```
int remote_size() const;
```

Returns the size of the remote group, i.e., the number of processes that are part of the remote group.

5.
```
boost::mpi::group remote_group() const;
```

Returns the remote group, containing all of the remote processes in this intercommunicator.

6.
```
communicator merge(bool high) const;
```

Merge the local and remote groups in this intercommunicator into a new intracommunicator containing the union of the processes in both groups. This method is equivalent to `MPI_Intercomm_merge`.

Parameters:		
	`high`	Whether the processes in this group should have the higher rank numbers than the processes in the other group. Each of the processes within a particular group shall have the same "high" value.
Returns:		the new, merged intracommunicator

Header <boost/mpi/nonblocking.hpp>

This header defines operations for completing non-blocking communication requests.

```
namespace boost {
  namespace mpi {
    template<typename ForwardIterator>
      std::pair< status, ForwardIterator >
      wait_any(ForwardIterator, ForwardIterator);
    template<typename ForwardIterator>
      optional< std::pair< status, ForwardIterator > >
      test_any(ForwardIterator, ForwardIterator);
    template<typename ForwardIterator, typename OutputIterator>
      OutputIterator
      wait_all(ForwardIterator, ForwardIterator, OutputIterator);
    template<typename ForwardIterator>
      void wait_all(ForwardIterator, ForwardIterator);
    template<typename ForwardIterator, typename OutputIterator>
      optional< OutputIterator >
      test_all(ForwardIterator, ForwardIterator, OutputIterator);
    template<typename ForwardIterator>
      bool test_all(ForwardIterator, ForwardIterator);
    template<typename BidirectionalIterator,
             typename OutputIterator>
      std::pair< OutputIterator, BidirectionalIterator >
      wait_some(BidirectionalIterator, BidirectionalIterator,
                OutputIterator);
    template<typename BidirectionalIterator>
      BidirectionalIterator
      wait_some(BidirectionalIterator, BidirectionalIterator);
    template<typename BidirectionalIterator,
             typename OutputIterator>
      std::pair< OutputIterator, BidirectionalIterator >
      test_some(BidirectionalIterator, BidirectionalIterator,
                OutputIterator);
    template<typename BidirectionalIterator>
      BidirectionalIterator
      test_some(BidirectionalIterator, BidirectionalIterator);
  }
}
```

Function template wait_any

boost::mpi::wait_any — Wait until any non-blocking request has completed.

Synopsis

```
// In header: <boost/mpi/nonblocking.hpp>


template<typename ForwardIterator>
  std::pair< status, ForwardIterator >
  wait_any(ForwardIterator first, ForwardIterator last);
```

Description

This routine takes in a set of requests stored in the iterator range [first,last) and waits until any of these requests has been completed. It provides functionality equivalent to `MPI_Waitany`.

Parameters: `first` The iterator that denotes the beginning of the sequence of request objects.
`last` The iterator that denotes the end of the sequence of request objects. This may not be equal to `first`.

Returns: A pair containing the status object that corresponds to the completed operation and the iterator referencing the completed request.

Function template test_any

boost::mpi::test_any — Test whether any non-blocking request has completed.

Synopsis

```
// In header: <boost/mpi/nonblocking.hpp>


template<typename ForwardIterator>
  optional< std::pair< status, ForwardIterator > >
  test_any(ForwardIterator first, ForwardIterator last);
```

Description

This routine takes in a set of requests stored in the iterator range [first,last) and tests whether any of these requests has been completed. This routine is similar to `wait_any`, but will not block waiting for requests to completed. It provides functionality equivalent to `MPI_Testany`.

Parameters:	`first`	The iterator that denotes the beginning of the sequence of request objects.
	`last`	The iterator that denotes the end of the sequence of request objects.
Returns:	If any outstanding requests have completed, a pair containing the status object that corresponds to the completed operation and the iterator referencing the completed request. Otherwise, an empty `optional<>`.	

Function wait_all

boost::mpi::wait_all — Wait until all non-blocking requests have completed.

Synopsis

```
// In header: <boost/mpi/nonblocking.hpp>


template<typename ForwardIterator, typename OutputIterator>
  OutputIterator
  wait_all(ForwardIterator first, ForwardIterator last,
           OutputIterator out);
template<typename ForwardIterator>
  void wait_all(ForwardIterator first, ForwardIterator last);
```

Description

This routine takes in a set of requests stored in the iterator range [first,last) and waits until all of these requests have been completed. It provides functionality equivalent to `MPI_Waitall`.

Parameters:	`first`	The iterator that denotes the beginning of the sequence of request objects.
	`last`	The iterator that denotes the end of the sequence of request objects.
	`out`	If provided, an output iterator through which the status of each request will be emitted. The `status` objects are emitted in the same order as the requests are retrieved from [first,last).
Returns:	If an `out` parameter was provided, the value `out` after all of the `status` objects have been emitted.	

Function test_all

boost::mpi::test_all — Tests whether all non-blocking requests have completed.

Synopsis

```
// In header: <boost/mpi/nonblocking.hpp>


template<typename ForwardIterator, typename OutputIterator>
  optional< OutputIterator >
  test_all(ForwardIterator first, ForwardIterator last,
           OutputIterator out);
template<typename ForwardIterator>
  bool test_all(ForwardIterator first, ForwardIterator last);
```

Description

This routine takes in a set of requests stored in the iterator range [first,last) and determines whether all of these requests have been completed. However, due to limitations of the underlying MPI implementation, if any of the requests refers to a non-blocking send or receive of a serialized data type, `test_all` will always return the equivalent of `false` (i.e., the requests cannot all be finished at this time). This routine performs the same functionality as `wait_all`, except that this routine will not block. This routine provides functionality equivalent to `MPI_Testall`.

Parameters:

`first`	The iterator that denotes the beginning of the sequence of request objects.
`last`	The iterator that denotes the end of the sequence of request objects.
`out`	If provided and all requests hav been completed, an output iterator through which the status of each request will be emitted. The `status` objects are emitted in the same order as the requests are retrieved from [first,last).

Returns: If an `out` parameter was provided, the value `out` after all of the `status` objects have been emitted (if all requests were completed) or an empty `optional<>`. If no `out` parameter was provided, returns `true` if all requests have completed or `false` otherwise.

Function wait_some

boost::mpi::wait_some — Wait until some non-blocking requests have completed.

Synopsis

```
// In header: <boost/mpi/nonblocking.hpp>


template<typename BidirectionalIterator, typename OutputIterator>
  std::pair< OutputIterator, BidirectionalIterator >
  wait_some(BidirectionalIterator first,
            BidirectionalIterator last, OutputIterator out);
template<typename BidirectionalIterator>
  BidirectionalIterator
  wait_some(BidirectionalIterator first,
            BidirectionalIterator last);
```

Description

This routine takes in a set of requests stored in the iterator range [first,last) and waits until at least one of the requests has completed. It then completes all of the requests it can, partitioning the input sequence into pending requests followed by completed requests. If an output iterator is provided, `status` objects will be emitted for each of the completed requests. This routine provides functionality equivalent to `MPI_Waitsome`.

Parameters:	`first`	The iterator that denotes the beginning of the sequence of request objects.
	`last`	The iterator that denotes the end of the sequence of request objects. This may not be equal to `first`.
	`out`	If provided, the `status` objects corresponding to completed requests will be emitted through this output iterator.
Returns:	If the `out` parameter was provided, a pair containing the output iterator `out` after all of the `status` objects have been written through it and an iterator referencing the first completed request. If no `out` parameter was provided, only the iterator referencing the first completed request will be emitted.	

Function test_some

boost::mpi::test_some — Test whether some non-blocking requests have completed.

Synopsis

```
// In header: <boost/mpi/nonblocking.hpp>


template<typename BidirectionalIterator, typename OutputIterator>
  std::pair< OutputIterator, BidirectionalIterator >
  test_some(BidirectionalIterator first,
            BidirectionalIterator last, OutputIterator out);
template<typename BidirectionalIterator>
  BidirectionalIterator
  test_some(BidirectionalIterator first,
            BidirectionalIterator last);
```

Description

This routine takes in a set of requests stored in the iterator range [first,last) and tests to see if any of the requests has completed. It completes all of the requests it can, partitioning the input sequence into pending requests followed by completed requests. If an output iterator is provided, `status` objects will be emitted for each of the completed requests. This routine is similar to `wait_some`, but does not wait until any requests have completed. This routine provides functionality equivalent to `MPI_Testsome`.

Parameters:	`first`	The iterator that denotes the beginning of the sequence of request objects.
	`last`	The iterator that denotes the end of the sequence of request objects. This may not be equal to `first`.
	`out`	If provided, the `status` objects corresponding to completed requests will be emitted through this output iterator.
Returns:	If the `out` parameter was provided, a pair containing the output iterator `out` after all of the `status` objects have been written through it and an iterator referencing the first completed request. If no `out` parameter was provided, only the iterator referencing the first completed request will be emitted.	

Header <boost/mpi/operations.hpp>

This header provides a mapping from function objects to `MPI_Op` constants used in MPI collective operations. It also provides several new function object types not present in the standard <functional> header that have direct mappings to `MPI_Op`.

```
namespace boost {
  namespace mpi {
    template<typename Op, typename T> struct is_commutative;
    template<typename T> struct maximum;
    template<typename T> struct minimum;
    template<typename T> struct bitwise_and;
    template<typename T> struct bitwise_or;
    template<typename T> struct logical_xor;
    template<typename T> struct bitwise_xor;
    template<typename Op, typename T> struct is_mpi_op;
  }
}
```

Struct template is_commutative

boost::mpi::is_commutative — Determine if a function object type is commutative.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename Op, typename T>
struct is_commutative {
};
```

Description

This trait determines if an operation `Op` is commutative when applied to values of type `T`. Parallel operations such as `reduce` and `prefix_sum` can be implemented more efficiently with commutative operations. To mark an operation as commutative, users should specialize `is_commutative` and derive from the class `mpl::true_`.

Struct template maximum

boost::mpi::maximum — Compute the maximum of two values.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename T>
struct maximum {

  // public member functions
  const T & operator()(const T &, const T &) const;
};
```

Description

This binary function object computes the maximum of the two values it is given. When used with MPI and a type `T` that has an associated, built-in MPI data type, translates to `MPI_MAX`.

`maximum` public member functions

1. ```
   const T & operator()(const T & x, const T & y) const;
   ```

   Returns: the maximum of x and y.
   ```

Struct template minimum

boost::mpi::minimum — Compute the minimum of two values.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename T>
struct minimum {

  // public member functions
  const T & operator()(const T &, const T &) const;
};
```

Description

This binary function object computes the minimum of the two values it is given. When used with MPI and a type `T` that has an associated, built-in MPI data type, translates to `MPI_MIN`.

`minimum` public member functions

1. ```
   const T & operator()(const T & x, const T & y) const;
   ```

   Returns: the minimum of x and y.
   ```

Struct template bitwise_and

boost::mpi::bitwise_and — Compute the bitwise AND of two integral values.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename T>
struct bitwise_and {

  // public member functions
  T operator()(const T &, const T &) const;
};
```

Description

This binary function object computes the bitwise AND of the two values it is given. When used with MPI and a type `T` that has an associated, built-in MPI data type, translates to `MPI_BAND`.

`bitwise_and` public member functions

1.
```
T operator()(const T & x, const T & y) const;
```

   Returns: x & y.

Struct template bitwise_or

boost::mpi::bitwise_or — Compute the bitwise OR of two integral values.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename T>
struct bitwise_or {

  // public member functions
  T operator()(const T &, const T &) const;
};
```

Description

This binary function object computes the bitwise OR of the two values it is given. When used with MPI and a type `T` that has an associated, built-in MPI data type, translates to `MPI_BOR`.

`bitwise_or` public member functions

1. ```
   T operator()(const T & x, const T & y) const;
   ```

   Returns: the x | y.
   ```

Struct template logical_xor

boost::mpi::logical_xor — Compute the logical exclusive OR of two integral values.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename T>
struct logical_xor {

  // public member functions
  T operator()(const T &, const T &) const;
};
```

Description

This binary function object computes the logical exclusive of the two values it is given. When used with MPI and a type `T` that has an associated, built-in MPI data type, translates to `MPI_LXOR`.

`logical_xor` public member functions

1.
```
T operator()(const T & x, const T & y) const;
```

   Returns: the logical exclusive OR of x and y.

Struct template bitwise_xor

boost::mpi::bitwise_xor — Compute the bitwise exclusive OR of two integral values.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename T>
struct bitwise_xor {

  // public member functions
  T operator()(const T &, const T &) const;
};
```

Description

This binary function object computes the bitwise exclusive OR of the two values it is given. When used with MPI and a type `T` that has an associated, built-in MPI data type, translates to `MPI_BXOR`.

`bitwise_xor` public member functions

1. ```
   T operator()(const T & x, const T & y) const;
   ```

   Returns: x ^ y.
   ```

Struct template is_mpi_op

boost::mpi::is_mpi_op — Determine if a function object has an associated `MPI_Op`.

Synopsis

```
// In header: <boost/mpi/operations.hpp>

template<typename Op, typename T>
struct is_mpi_op {
};
```

Description

This trait determines if a function object type `Op`, when used with argument type `T`, has an associated `MPI_Op`. If so, `is_mpi_op<Op,T>` will derive from `mpl::false_` and will contain a static member function `op` that takes no arguments but returns the associated `MPI_Op` value. For instance, `is_mpi_op<std::plus<int>,int>op()` returns `MPI_SUM`.

Users may specialize `is_mpi_op` for any other class templates that map onto operations that have `MPI_Op` equivalences, such as bitwise OR, logical and, or maximum. However, users are encouraged to use the standard function objects in the `functional` and `boost/mpi/operations.hpp` headers whenever possible. For function objects that are class templates with a single template parameter, it may be easier to specialize `is_builtin_mpi_op`.

Header <boost/mpi/packed_iarchive.hpp>

This header provides the facilities for packing Serializable data types into a buffer using `MPI_Pack`. The buffers can then be transmitted via MPI and then be unpacked either via the facilities in `packed_oarchive.hpp` or `MPI_Unpack`.

```
namespace boost {
  namespace mpi {
    class packed_iarchive;

    typedef packed_iprimitive iprimitive;
  }
}
```

Class packed_iarchive

boost::mpi::packed_iarchive — An archive that packs binary data into an MPI buffer.

Synopsis

```
// In header: <boost/mpi/packed_iarchive.hpp>


class packed_iarchive {
public:
  // construct/copy/destruct
  packed_iarchive(MPI_Comm const &, buffer_type &,
                  unsigned int = boost::archive::no_header,
                  int = 0);
  packed_iarchive(MPI_Comm const &, std::size_t = 0,
                  unsigned int = boost::archive::no_header);

  // public member functions
  template<typename T> void load_override(T &, int, mpl::false_);
  template<typename T> void load_override(T &, int, mpl::true_);
  template<typename T> void load_override(T &, int);
  void load_override(archive::class_id_optional_type &, int);
  void load_override(archive::class_name_type &, int);
};
```

Description

The `packed_iarchive` class is an Archiver (as in the Boost.Serialization library) that packs binary data into a buffer for transmission via MPI. It can operate on any Serializable data type and will use the `MPI_Pack` function of the underlying MPI implementation to perform serialization.

`packed_iarchive` public construct/copy/destruct

1.
```
packed_iarchive(MPI_Comm const & comm, buffer_type & b,
                unsigned int flags = boost::archive::no_header,
                int position = 0);
```

Construct a `packed_iarchive` for transmission over the given MPI communicator and with an initial buffer.

Parameters:		
	`b`	A user-defined buffer that will be filled with the binary representation of serialized objects.
	`comm`	The communicator over which this archive will be sent.
	`flags`	Control the serialization of the data types. Refer to the Boost.Serialization documentation before changing the default flags.
	`position`	Set the offset into buffer `b` at which deserialization will begin.

2.
```
packed_iarchive(MPI_Comm const & comm, std::size_t s = 0,
                unsigned int flags = boost::archive::no_header);
```

Construct a `packed_iarchive` for transmission over the given MPI communicator.

Parameters:		
	`comm`	The communicator over which this archive will be sent.
	`flags`	Control the serialization of the data types. Refer to the Boost.Serialization documentation before changing the default flags.
	`s`	The size of the buffer to be received.

`packed_iarchive` public member functions

1.
```
template<typename T>
  void load_override(T & x, int version, mpl::false_);
```

2.
```
template<typename T> void load_override(T & x, int, mpl::true_);
```

3.
```
template<typename T> void load_override(T & x, int version);
```

4.
```
void load_override(archive::class_id_optional_type &, int);
```

5.
```
void load_override(archive::class_name_type & t, int);
```

Header <boost/mpi/packed_oarchive.hpp>

This header provides the facilities for unpacking Serializable data types from a buffer using `MPI_Unpack`. The buffers are typically received via MPI and have been packed either by via the facilities in `packed_iarchive.hpp` or `MPI_Pack`.

```
namespace boost {
  namespace mpi {
    class packed_oarchive;

    typedef packed_oprimitive oprimitive;
  }
}
```

Class packed_oarchive

boost::mpi::packed_oarchive — An archive that unpacks binary data from an MPI buffer.

Synopsis

```
// In header: <boost/mpi/packed_oarchive.hpp>


class packed_oarchive {
public:
  // construct/copy/destruct
  packed_oarchive(MPI_Comm const &, buffer_type &,
                  unsigned int = boost::archive::no_header);
  packed_oarchive(MPI_Comm const &,
                  unsigned int = boost::archive::no_header);

  // public member functions
  template<typename T>
    void save_override(T const &, int, mpl::false_);
  template<typename T>
    void save_override(T const &, int, mpl::true_);
  template<typename T> void save_override(T const &, int);
  void save_override(const archive::class_id_optional_type &, int);
  void save_override(const archive::class_name_type &, int);
};
```

Description

The `packed_oarchive` class is an Archiver (as in the Boost.Serialization library) that unpacks binary data from a buffer received via MPI. It can operate on any Serializable data type and will use the `MPI_Unpack` function of the underlying MPI implementation to perform deserialization.

`packed_oarchive` public construct/copy/destruct

1.
```
packed_oarchive(MPI_Comm const & comm, buffer_type & b,
                unsigned int flags = boost::archive::no_header);
```

Construct a `packed_oarchive` to receive data over the given MPI communicator and with an initial buffer.

Parameters:

`b`	A user-defined buffer that contains the binary representation of serialized objects.
`comm`	The communicator over which this archive will be received.
`flags`	Control the serialization of the data types. Refer to the Boost.Serialization documentation before changing the default flags.

2.
```
packed_oarchive(MPI_Comm const & comm,
                unsigned int flags = boost::archive::no_header);
```

Construct a `packed_oarchive` to receive data over the given MPI communicator.

Parameters:

`comm`	The communicator over which this archive will be received.
`flags`	Control the serialization of the data types. Refer to the Boost.Serialization documentation before changing the default flags.

`packed_oarchive` public member functions

1.
```
template<typename T>
  void save_override(T const & x, int version, mpl::false_);
```

2.
```
template<typename T>
  void save_override(T const & x, int, mpl::true_);
```

3.
```
template<typename T> void save_override(T const & x, int version);
```

4.
```
void save_override(const archive::class_id_optional_type &, int);
```

5.
```
void save_override(const archive::class_name_type & t, int);
```

Header <boost/mpi/python.hpp>

This header interacts with the Python bindings for Boost.MPI. The routines in this header can be used to register user-defined and library-defined data types with Boost.MPI for efficient (de-)serialization and separate transmission of skeletons and content.

```
namespace boost {
  namespace mpi {
    namespace python {
      template<typename T>
        void register_serialized(const T & = T(),
                                 PyTypeObject * = 0);
      template<typename T>
        void register_skeleton_and_content(const T & = T(),
                                           PyTypeObject * = 0);
    }
  }
}
```

Function template register_serialized

boost::mpi::python::register_serialized — Register the type T for direct serialization within Boost.MPI.

Synopsis

```
// In header: <boost/mpi/python.hpp>


template<typename T>
  void register_serialized(const T & value = T(),
                           PyTypeObject * type = 0);
```

Description

The `register_serialized` function registers a C++ type for direct serialization within Boost.MPI. Direct serialization elides the use of the Python `pickle` package when serializing Python objects that represent C++ values. Direct serialization can be beneficial both to improve serialization performance (Python pickling can be very inefficient) and to permit serialization for Python-wrapped C++ objects that do not support pickling.

Parameters:

`type`	The Python type associated with the C++ type `T`. If not provided, it will be computed from the same value `value`.
`value`	A sample value of the type `T`. This may be used to compute the Python type associated with the C++ type `T`.

Function template register_skeleton_and_content

boost::mpi::python::register_skeleton_and_content — Registers a type for use with the skeleton/content mechanism in Python.

Synopsis

```
// In header: <boost/mpi/python.hpp>


template<typename T>
  void register_skeleton_and_content(const T & value = T(),
                                     PyTypeObject * type = 0);
```

Description

The skeleton/content mechanism can only be used from Python with C++ types that have previously been registered via a call to this function. Both the sender and the transmitter must register the type. It is permitted to call this function multiple times for the same type `T`, but only one call per process per type is required. The type `T` must be Serializable.

Parameters:

- `type` The Python type associated with the C++ type `T`. If not provided, it will be computed from the same value `value`.
- `value` A sample object of type T that will be used to determine the Python type associated with T, if `type` is not specified.

Header <boost/mpi/request.hpp>

This header defines the class `request`, which contains a request for non-blocking communication.

```
namespace boost {
  namespace mpi {
    class request;
  }
}
```

Class request

boost::mpi::request — A request for a non-blocking send or receive.

Synopsis

```
// In header: <boost/mpi/request.hpp>


class request {
public:
  // construct/copy/destruct
  request();

  // public member functions
  status wait();
  optional< status > test();
  void cancel();
};
```

Description

This structure contains information about a non-blocking send or receive and will be returned from `isend` or `irecv`, respectively.

`request` public construct/copy/destruct

1. ```
   request();
   ```

   Constructs a NULL request.

`request` public member functions

1. ```
   status wait();
   ```

   Wait until the communication associated with this request has completed, then return a `status` object describing the communication.

2. ```
   optional< status > test();
   ```

   Determine whether the communication associated with this request has completed successfully. If so, returns the `status` object describing the communication. Otherwise, returns an empty `optional<>` to indicate that the communication has not completed yet. Note that once `test()` returns a `status` object, the request has completed and `wait()` should not be called.

3. ```
   void cancel();
   ```

   Cancel a pending communication, assuming it has not already been completed.

Header <boost/mpi/skeleton_and_content.hpp>

This header provides facilities that allow the structure of data types (called the "skeleton") to be transmitted and received separately from the content stored in those data types. These facilities are useful when the data in a stable data structure (e.g., a mesh or a graph) will need to be transmitted repeatedly. In this case, transmitting the skeleton only once saves both communication effort (it need not be sent again) and local computation (serialization need only be performed once for the content).

```
namespace boost {
  namespace mpi {
    template<typename T> struct skeleton_proxy;

    class content;
    class packed_skeleton_iarchive;
    class packed_skeleton_oarchive;
    template<typename T> const skeleton_proxy< T > skeleton(T &);
    template<typename T> const content get_content(const T &);
  }
}
```

Struct template skeleton_proxy

boost::mpi::skeleton_proxy — A proxy that requests that the skeleton of an object be transmitted.

Synopsis

```
// In header: <boost/mpi/skeleton_and_content.hpp>

template<typename T>
struct skeleton_proxy {
  // construct/copy/destruct
  skeleton_proxy(T &);
  T & object;
};
```

Description

The `skeleton_proxy` is a lightweight proxy object used to indicate that the skeleton of an object, not the object itself, should be transmitted. It can be used with the `send` and `recv` operations of communicators or the `broadcast` collective. When a `skeleton_proxy` is sent, Boost.MPI generates a description containing the structure of the stored object. When that skeleton is received, the receiving object is reshaped to match the structure. Once the skeleton of an object as been transmitted, its `content` can be transmitted separately (often several times) without changing the structure of the object.

`skeleton_proxy` public construct/copy/destruct

1. ```
   skeleton_proxy(T & x);
   ```

   Constructs a `skeleton_proxy` that references object `x`.

   Parameters: `x` the object whose structure will be transmitted or altered.
   ```

Class content

boost::mpi::content — A proxy object that transfers the content of an object without its structure.

Synopsis

```
// In header: <boost/mpi/skeleton_and_content.hpp>


class content {
public:
  // construct/copy/destruct
  content();
  content(MPI_Datatype, bool = true);
  content& operator=(MPI_Datatype);

  // public member functions
  MPI_Datatype get_mpi_datatype() const;
  void commit();
};
```

Description

The `content` class indicates that Boost.MPI should transmit or receive the content of an object, but without any information about the structure of the object. It is only meaningful to transmit the content of an object after the receiver has already received the skeleton for the same object.

Most users will not use `content` objects directly. Rather, they will invoke `send`, `recv`, or `broadcast` operations using `get_content()`.

`content` public construct/copy/destruct

1. ```
   content();
   ```

   Constructs an empty `content` object. This object will not be useful for any Boost.MPI operations until it is reassigned.

2. ```
   content(MPI_Datatype d, bool committed = true);
   ```

   This routine initializes the `content` object with an MPI data type that refers to the content of an object without its structure.

   Parameters: `committed` `true` indicates that `MPI_Type_commit` has already been excuted for the data type `d`.
   `d` the MPI data type referring to the content of the object.

3. ```
   content& operator=(MPI_Datatype d);
   ```

   Replace the MPI data type referencing the content of an object.

   Parameters: `d` the new MPI data type referring to the content of the object.
   Returns: *this

`content` public member functions

1. ```
   MPI_Datatype get_mpi_datatype() const;
   ```

   Retrieve the MPI data type that refers to the content of the object.

Returns: the MPI data type, which should only be transmitted or received using `MPI_BOTTOM` as the address.

2.
```
void commit();
```

Commit the MPI data type referring to the content of the object.

Class packed_skeleton_iarchive

boost::mpi::packed_skeleton_iarchive — An archiver that reconstructs a data structure based on the binary skeleton stored in a buffer.

Synopsis

```
// In header: <boost/mpi/skeleton_and_content.hpp>


class packed_skeleton_iarchive {
public:
  // construct/copy/destruct
  packed_skeleton_iarchive(MPI_Comm const &,
                           unsigned int = boost::archive::no_header);
  packed_skeleton_iarchive(packed_iarchive &);

  // public member functions
  const packed_iarchive & get_skeleton() const;
  packed_iarchive & get_skeleton();
};
```

Description

The `packed_skeleton_iarchive` class is an Archiver (as in the Boost.Serialization library) that can construct the the shape of a data structure based on a binary skeleton stored in a buffer. The `packed_skeleton_iarchive` is typically used by the receiver of a skeleton, to prepare a data structure that will eventually receive content separately.

Users will not generally need to use `packed_skeleton_iarchive` directly. Instead, use `skeleton` or `get_skeleton`.

`packed_skeleton_iarchive` public construct/copy/destruct

1.
```
packed_skeleton_iarchive(MPI_Comm const & comm,
                         unsigned int flags = boost::archive::no_header);
```

Construct a `packed_skeleton_iarchive` for the given communicator.

Parameters:
- `comm` The communicator over which this archive will be transmitted.
- `flags` Control the serialization of the skeleton. Refer to the Boost.Serialization documentation before changing the default flags.

2.
```
packed_skeleton_iarchive(packed_iarchive & archive);
```

Construct a `packed_skeleton_iarchive` that unpacks a skeleton from the given `archive`.

Parameters:
- `archive` the archive from which the skeleton will be unpacked.

`packed_skeleton_iarchive` public member functions

1.
```
const packed_iarchive & get_skeleton() const;
```

Retrieve the archive corresponding to this skeleton.

2.
```
packed_iarchive & get_skeleton();
```

Retrieve the archive corresponding to this skeleton.

Class packed_skeleton_oarchive

boost::mpi::packed_skeleton_oarchive — An archiver that records the binary skeleton of a data structure into a buffer.

Synopsis

```
// In header: <boost/mpi/skeleton_and_content.hpp>


class packed_skeleton_oarchive {
public:
  // construct/copy/destruct
  packed_skeleton_oarchive(MPI_Comm const &,
                           unsigned int = boost::archive::no_header);
  packed_skeleton_oarchive(packed_oarchive &);

  // public member functions
  const packed_oarchive & get_skeleton() const;
};
```

Description

The `packed_skeleton_oarchive` class is an Archiver (as in the Boost.Serialization library) that can record the shape of a data structure (called the "skeleton") into a binary representation stored in a buffer. The `packed_skeleton_oarchive` is typically used by the send of a skeleton, to pack the skeleton of a data structure for transmission separately from the content.

Users will not generally need to use `packed_skeleton_oarchive` directly. Instead, use `skeleton` or `get_skeleton`.

`packed_skeleton_oarchive` public construct/copy/destruct

1.
```
packed_skeleton_oarchive(MPI_Comm const & comm,
                         unsigned int flags = boost::archive::no_header);
```

Construct a `packed_skeleton_oarchive` for the given communicator.

Parameters: `comm` The communicator over which this archive will be transmitted.
`flags` Control the serialization of the skeleton. Refer to the Boost.Serialization documentation before changing the default flags.

2.
```
packed_skeleton_oarchive(packed_oarchive & archive);
```

Construct a `packed_skeleton_oarchive` that packs a skeleton into the given `archive`.

Parameters: `archive` the archive to which the skeleton will be packed.

`packed_skeleton_oarchive` public member functions

1.
```
const packed_oarchive & get_skeleton() const;
```

Retrieve the archive corresponding to this skeleton.

Function template skeleton

boost::mpi::skeleton — Create a skeleton proxy object.

Synopsis

```
// In header: <boost/mpi/skeleton_and_content.hpp>

template<typename T> const skeleton_proxy< T > skeleton(T & x);
```

Description

This routine creates an instance of the skeleton_proxy class. It will typically be used when calling `send`, `recv`, or `broadcast`, to indicate that only the skeleton (structure) of an object should be transmitted and not its contents.

Parameters: `x` the object whose structure will be transmitted.
Returns: a skeleton_proxy object referencing `x`

Function template get_content

boost::mpi::get_content — Returns the content of an object, suitable for transmission via Boost.MPI.

Synopsis

```
// In header: <boost/mpi/skeleton_and_content.hpp>


template<typename T> const content get_content(const T & x);
```

Description

The function creates an absolute MPI datatype for the object, where all offsets are counted from the address 0 (a.k.a. `MPI_BOTTOM`) instead of the address `&x` of the object. This allows the creation of MPI data types for complex data structures containing pointers, such as linked lists or trees.

The disadvantage, compared to relative MPI data types is that for each object a new MPI data type has to be created.

The contents of an object can only be transmitted when the receiver already has an object with the same structure or shape as the sender. To accomplish this, first transmit the skeleton of the object using, e.g., `skeleton()` or `skeleton_proxy`.

The type `T` has to allow creation of an absolute MPI data type (content).

Parameters: `x` the object for which the content will be transmitted.
Returns: the content of the object `x`, which can be used for transmission via `send`, `recv`, or `broadcast`.

Header <boost/mpi/skeleton_and_content_fwd.hpp>

This header contains all of the forward declarations required to use transmit skeletons of data structures and the content of data structures separately. To actually transmit skeletons or content, include the header `boost/mpi/skeleton_and_content.hpp`.

Header <boost/mpi/status.hpp>

This header defines the class `status`, which reports on the results of point-to-point communication.

```
namespace boost {
  namespace mpi {
    class status;
  }
}
```

Class status

boost::mpi::status — Contains information about a message that has been or can be received.

Synopsis

```
// In header: <boost/mpi/status.hpp>


class status {
public:
  // construct/copy/destruct
  status();
  status(MPI_Status const &);

  // public member functions
  int source() const;
  int tag() const;
  int error() const;
  bool cancelled() const;
  template<typename T> optional< int > count() const;
  operator MPI_Status &();
  operator const MPI_Status &() const;
  mutable int m_count;
};
```

Description

This structure contains status information about messages that have been received (with `communicator::recv`) or can be received (returned from `communicator::probe` or `communicator::iprobe`). It permits access to the source of the message, message tag, error code (rarely used), or the number of elements that have been transmitted.

`status` public construct/copy/destruct

1.
```
status();
```

2.
```
status(MPI_Status const & s);
```

`status` public member functions

1.
```
int source() const;
```

Retrieve the source of the message.

2.
```
int tag() const;
```

Retrieve the message tag.

3.
```
int error() const;
```

Retrieve the error code.

4.
```
bool cancelled() const;
```

Determine whether the communication associated with this object has been successfully cancelled.

5.
```
template<typename T> optional< int > count() const;
```

Determines the number of elements of type `T` contained in the message. The type `T` must have an associated data type, i.e., `is_mpi_datatype<T>` must derive `mpl::true_`. In cases where the type `T` does not match the transmitted type, this routine will return an empty `optional<int>`.

Returns: the number of `T` elements in the message, if it can be determined.

6.
```
operator MPI_Status &();
```

References the underlying `MPI_Status`

7.
```
operator const MPI_Status &() const;
```

References the underlying `MPI_Status`

Header <boost/mpi/timer.hpp>

This header provides the `timer` class, which provides access to the MPI timers.

```
namespace boost {
  namespace mpi {
    class timer;
  }
}
```

Class timer

boost::mpi::timer — A simple timer that provides access to the MPI timing facilities.

Synopsis

```
// In header: <boost/mpi/timer.hpp>


class timer {
public:
  // construct/copy/destruct
  timer();

  // public member functions
  void restart();
  double elapsed() const;
  double elapsed_max() const;
  double elapsed_min() const;

  // public static functions
  static bool time_is_global();
};
```

Description

The `timer` class is a simple wrapper around the MPI timing facilities that mimics the interface of the Boost Timer library.

`timer` public construct/copy/destruct

1. ```
   timer();
   ```

   Initializes the timer

   Postconditions: `elapsed()` == 0

`timer` public member functions

1. ```
   void restart();
   ```

   Restart the timer.

   Postconditions: `elapsed()` == 0

2. ```
   double elapsed() const;
   ```

   Return the amount of time that has elapsed since the last construction or reset, in seconds.

3. ```
   double elapsed_max() const;
   ```

   Return an estimate of the maximum possible value of elapsed(). Note that this routine may return too high a value on some systems.

4. ```
   double elapsed_min() const;
   ```

   Returns the minimum non-zero value that `elapsed()` may return. This is the resolution of the timer.
   ```

`timer` public static functions

1.
```
static bool time_is_global();
```

Determines whether the elapsed time values are global times or local processor times.

Python Bindings

Boost.MPI provides an alternative MPI interface from the Python programming language via the `boost.mpi` module. The Boost.MPI Python bindings, built on top of the C++ Boost.MPI using the Boost.Python library, provide nearly all of the functionality of Boost.MPI within a dynamic, object-oriented language.

The Boost.MPI Python module can be built and installed from the `libs/mpi/build` directory. Just follow the configuration and installation instructions for the C++ Boost.MPI. Once you have installed the Python module, be sure that the installation location is in your `PYTHONPATH`.

Quickstart

Getting started with the Boost.MPI Python module is as easy as importing `boost.mpi`. Our first "Hello, World!" program is just two lines long:

```
import boost.mpi as mpi
print "I am process %d of %d." % (mpi.rank, mpi.size)
```

Go ahead and run this program with several processes. Be sure to invoke the `python` interpreter from `mpirun`, e.g.,

```
mpirun -np 5 python hello_world.py
```

This will return output such as:

```
I am process 1 of 5.
I am process 3 of 5.
I am process 2 of 5.
I am process 4 of 5.
I am process 0 of 5.
```

Point-to-point operations in Boost.MPI have nearly the same syntax in Python as in C++. We can write a simple two-process Python program that prints "Hello, world!" by transmitting Python strings:

```
import boost.mpi as mpi

if mpi.world.rank == 0:
  mpi.world.send(1, 0, 'Hello')
  msg = mpi.world.recv(1, 1)
  print msg,'!'
else:
  msg = mpi.world.recv(0, 0)
  print (msg + ', '),
  mpi.world.send(0, 1, 'world')
```

There are only a few notable differences between this Python code and the example in the C++ tutorial. First of all, we don't need to write any initialization code in Python: just loading the `boost.mpi` module makes the appropriate `MPI_Init` and `MPI_Finalize` calls. Second, we're passing Python objects from one process to another through MPI. Any Python object that can be pickled can be transmitted; the next section will describe in more detail how the Boost.MPI Python layer transmits objects. Finally, when we receive objects with `recv`, we don't need to specify the type because transmission of Python objects is polymorphic.

When experimenting with Boost.MPI in Python, don't forget that help is always available via `pydoc`: just pass the name of the module or module entity on the command line (e.g., `pydoc boost.mpi.communicator`) to receive complete reference documentation. When in doubt, try it!

Transmitting User-Defined Data

Boost.MPI can transmit user-defined data in several different ways. Most importantly, it can transmit arbitrary Python objects by pickling them at the sender and unpickling them at the receiver, allowing arbitrarily complex Python data structures to interoperate with MPI.

Boost.MPI also supports efficient serialization and transmission of C++ objects (that have been exposed to Python) through its C++ interface. Any C++ type that provides (de-)serialization routines that meet the requirements of the Boost.Serialization library is eligible for this optimization, but the type must be registered in advance. To register a C++ type, invoke the C++ function `register_serialized`. If your C++ types come from other Python modules (they probably will!), those modules will need to link against the `boost_mpi` and `boost_mpi_python` libraries as described in the installation section. Note that you do **not** need to link against the Boost.MPI Python extension module.

Finally, Boost.MPI supports separation of the structure of an object from the data it stores, allowing the two pieces to be transmitted separately. This "skeleton/content" mechanism, described in more detail in a later section, is a communication optimization suitable for problems with fixed data structures whose internal data changes frequently.

Collectives

Boost.MPI supports all of the MPI collectives (`scatter`, `reduce`, `scan`, `broadcast`, etc.) for any type of data that can be transmitted with the point-to-point communication operations. For the MPI collectives that require a user-specified operation (e.g., `reduce` and `scan`), the operation can be an arbitrary Python function. For instance, one could concatenate strings with `all_reduce`:

```
mpi.all_reduce(my_string, lambda x,y: x + y)
```

The following module-level functions implement MPI collectives: all_gather Gather the values from all processes. all_reduce Combine the results from all processes. all_to_all Every process sends data to every other process. broadcast Broadcast data from one process to all other processes. gather Gather the values from all processes to the root. reduce Combine the results from all processes to the root. scan Prefix reduction of the values from all processes. scatter Scatter the values stored at the root to all processes.

Skeleton/Content Mechanism

Boost.MPI provides a skeleton/content mechanism that allows the transfer of large data structures to be split into two separate stages, with the skeleton (or, "shape") of the data structure sent first and the content (or, "data") of the data structure sent later, potentially several times, so long as the structure has not changed since the skeleton was transferred. The skeleton/content mechanism can improve performance when the data structure is large and its shape is fixed, because while the skeleton requires serialization (it has an unknown size), the content transfer is fixed-size and can be done without extra copies.

To use the skeleton/content mechanism from Python, you must first register the type of your data structure with the skeleton/content mechanism **from C++**. The registration function is `register_skeleton_and_content` and resides in the `<boost/mpi/python.hpp>` header.

Once you have registered your C++ data structures, you can extract the skeleton for an instance of that data structure with `skeleton()`. The resulting `skeleton_proxy` can be transmitted via the normal send routine, e.g.,

```
mpi.world.send(1, 0, skeleton(my_data_structure))
```

`skeleton_proxy` objects can be received on the other end via `recv()`, which stores a newly-created instance of your data structure with the same "shape" as the sender in its `"object` attribute:

```
shape = mpi.world.recv(0, 0)
my_data_structure = shape.object
```

Once the skeleton has been transmitted, the content (accessed via `get_content`) can be transmitted in much the same way. Note, however, that the receiver also specifies `get_content(my_data_structure)` in its call to receive:

```
if mpi.rank == 0:
  mpi.world.send(1, 0, get_content(my_data_structure))
else:
  mpi.world.recv(0, 0, get_content(my_data_structure))
```

Of course, this transmission of content can occur repeatedly, if the values in the data structure--but not its shape--changes.

The skeleton/content mechanism is a structured way to exploit the interaction between custom-built MPI datatypes and `MPI_BOTTOM`, to eliminate extra buffer copies.

C++/Python MPI Compatibility

Boost.MPI is a C++ library whose facilities have been exposed to Python via the Boost.Python library. Since the Boost.MPI Python bindings are build directly on top of the C++ library, and nearly every feature of C++ library is available in Python, hybrid C++/Python programs using Boost.MPI can interact, e.g., sending a value from Python but receiving that value in C++ (or vice versa). However, doing so requires some care. Because Python objects are dynamically typed, Boost.MPI transfers type information along with the serialized form of the object, so that the object can be received even when its type is not known. This mechanism differs from its C++ counterpart, where the static types of transmitted values are always known.

The only way to communicate between the C++ and Python views on Boost.MPI is to traffic entirely in Python objects. For Python, this is the normal state of affairs, so nothing will change. For C++, this means sending and receiving values of type `boost::python::object`, from the Boost.Python library. For instance, say we want to transmit an integer value from Python:

```
comm.send(1, 0, 17)
```

In C++, we would receive that value into a Python object and then `extract` an integer value:

```
boost::python::object value;
comm.recv(0, 0, value);
int int_value = boost::python::extract<int>(value);
```

In the future, Boost.MPI will be extended to allow improved interoperability with the C++ Boost.MPI and the C MPI bindings.

Reference

The Boost.MPI Python module, `boost.mpi`, has its own reference documentation, which is also available using `pydoc` (from the command line) or `help(boost.mpi)` (from the Python interpreter).

Design Philosophy

The design philosophy of the Parallel MPI library is very simple: be both convenient and efficient. MPI is a library built for high-performance applications, but it's FORTRAN-centric, performance-minded design makes it rather inflexible from the C++ point of view: passing a string from one process to another is inconvenient, requiring several messages and explicit buffering; passing a container of strings from one process to another requires an extra level of manual bookkeeping; and passing a map from strings to containers of strings is positively infuriating. The Parallel MPI library allows all of these data types to be passed using the same simple `send()` and `recv()` primitives. Likewise, collective operations such as `reduce()` allow arbitrary data types and function objects, much like the C++ Standard Library would.

The higher-level abstractions provided for convenience must not have an impact on the performance of the application. For instance, sending an integer via `send` must be as efficient as a call to `MPI_Send`, which means that it must be implemented by a simple call to `MPI_Send`; likewise, an integer `reduce()` using `std::plus<int>` must be implemented with a call to `MPI_Reduce` on integers using the `MPI_SUM` operation: anything less will impact performance. In essence, this is the "don't pay for what you don't use" principle: if the user is not transmitting strings, s/he should not pay the overhead associated with strings.

Sometimes, achieving maximal performance means foregoing convenient abstractions and implementing certain functionality using lower-level primitives. For this reason, it is always possible to extract enough information from the abstractions in Boost.MPI to minimize the amount of effort required to interface between Boost.MPI and the C MPI library.

Performance Evaluation

Message-passing performance is crucial in high-performance distributed computing. To evaluate the performance of Boost.MPI, we modified the standard NetPIPE benchmark (version 3.6.2) to use Boost.MPI and compared its performance against raw MPI. We ran five different variants of the NetPIPE benchmark:

1. MPI: The unmodified NetPIPE benchmark.

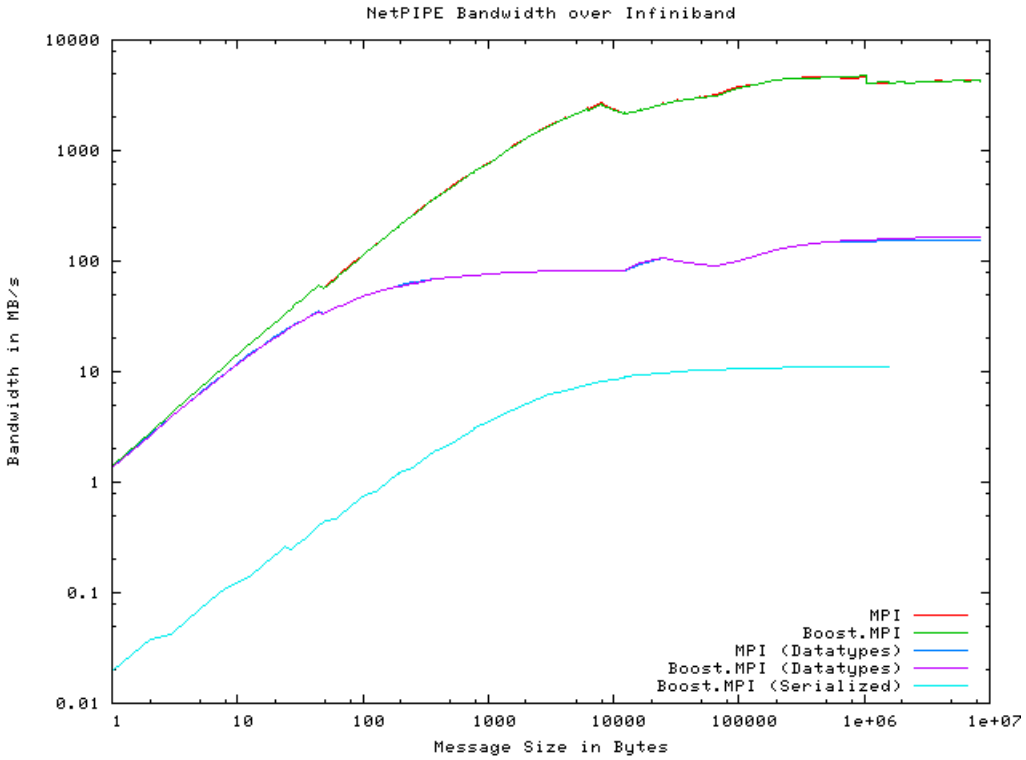
2. Boost.MPI: NetPIPE modified to use Boost.MPI calls for communication.

3. MPI (Datatypes): NetPIPE modified to use a derived datatype (which itself contains a single `MPI_BYTE`) rathan than a fundamental datatype.

4. Boost.MPI (Datatypes): NetPIPE modified to use a user-defined type `Char` in place of the fundamental `char` type. The `Char` type contains a single `char`, a `serialize()` method to make it serializable, and specializes `is_mpi_datatype` to force Boost.MPI to build a derived MPI data type for it.

5. Boost.MPI (Serialized): NetPIPE modified to use a user-defined type `Char` in place of the fundamental `char` type. This `Char` type contains a single `char` and is serializable. Unlike the Datatypes case, `is_mpi_datatype` is **not** specialized, forcing Boost.MPI to perform many, many serialization calls.

The actual tests were performed on the Odin cluster in the Department of Computer Science at Indiana University, which contains 128 nodes connected via Infiniband. Each node contains 4GB memory and two AMD Opteron processors. The NetPIPE benchmarks were compiled with Intel's C++ Compiler, version 9.0, Boost 1.35.0 (prerelease), and Open MPI version 1.1. The NetPIPE results follow:



There are a some observations we can make about these NetPIPE results. First of all, the top two plots show that Boost.MPI performs on par with MPI for fundamental types. The next two plots show that Boost.MPI performs on par with MPI for derived data types, even though Boost.MPI provides a much more abstract, completely transparent approach to building derived data types than raw MPI. Overall performance for derived data types is significantly worse than for fundamental data types, but the bottleneck is in the underlying MPI implementation itself. Finally, when forcing Boost.MPI to serialize characters individually, performance suffers greatly. This particular instance is the worst possible case for Boost.MPI, because we are serializing millions of individual characters. Overall, the additional abstraction provided by Boost.MPI does not impair its performance.

Revision History

- **Boost 1.36.0**:
  - Support for non-blocking operations in Python, from Andreas Klöckner
- **Boost 1.35.0**: Initial release, containing the following post-review changes
  - Support for arrays in all collective operations
  - Support default-construction of `environment`
- **2006-09-21**: Boost.MPI accepted into Boost.

Acknowledgments

Boost.MPI was developed with support from Zurcher Kantonalbank. Daniel Egloff and Michael Gauckler contributed many ideas to Boost.MPI's design, particularly in the design of its abstractions for MPI data types and the novel skeleton/context mechanism for large data structures. Prabhanjan (Anju) Kambadur developed the predecessor to Boost.MPI that proved the usefulness of the Serialization library in an MPI setting and the performance benefits of specialization in a C++ abstraction layer for MPI. Jeremy Siek managed the formal review of Boost.MPI.