Object Oriented MPI (OOMPI): A Class Library for the Message Passing Interface

Brian C. McCandless, Jeffrey M. Squyres, Andrew Lumsdaine
Laboratory for Scientific Computing
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
oompi@lsc.nd.edu

Abstract

Using the Message Passing Interface (MPI) in C++ has been difficult to this point because of the lack of suitable C++ bindings and C++ class libraries. The existing MPI standard provides language bindings only for C and Fortran 77, precluding their direct use in object oriented programming. Even the proposed C++ bindings in MPI-2 are at a fairly low-level and are not directly suitable for object-oriented programming. In this paper we present the requirements, analysis, and design for Object-Oriented MPI (OOMPI), a C++ class library for MPI. Although the OOMPI class library is specified in C++ , in some sense the specification is a generic one that uses C++ as the program description language. Thus, the OOMPI specification can also be considered as a generic object oriented class library specification which can thus also form the basis for MPI class libraries in other object-oriented languages.

1. Introduction

Although the Message Passing Interface (MPI) [3, 4, 9] is an increasingly popular and important standard for performance portable parallel programming, using MPI with C++ has been difficult because of the lack of standard C++ bindings and class libraries. There are three immediate alternatives for using MPI with C++: use the existing C bindings, use the proposed C++ bindings, or use a class library.

C Bindings. The C bindings for MPI provide one available interface for writing MPI programs in C++. However, making calls to the MPI C bindings from within a C++ program is cumbersome and such practice discards much of the expressive power of C++.

C++ Bindings. It has been proposed that the C++ interface to MPI should be a basic set of classes corresponding to the fundamental object types in MPI-1 (see [7]). This type of interface would be fairly lightweight. For instance, MPI error codes would still be returned by function calls, no new types of objects would be introduced, etc. Thus, only minimal use of advanced features of C++ such as polymorphism would be available to MPI programmers.

Class Library. A class library for MPI would make full use of C++ features such as inheritance, polymorphism, etc. For example, a class library would use polymorphism to obviate the need to explicitly provide type information of messages. By way of contrast, the C++ bindings would still require that type information be explicitly provided.

Note that a class library is not the same as a C++ binding. While a class library may make user programming more elegant, a binding must provide a direct and unambiguous mapping to the specified MPI functionality. In general, an MPI class library would be layered on the C or C++ bindings.

Other Work. MPI++ is one of the earliest proposed C++ interfaces to MPI [8]. MPI++ has as one if its goals to be semantically and syntactically consistent with the C interface. The basic design of MPI++ would thus form a sold basis for a C++ bindings for MPI.

mpi++ is a more recently introduced C++ interface to MPI [5]. The version described in [5] includes only point to point communication operations. However, even with these operations, mpi++ does not appear to be semantically or syntactically consistent with the C interface.

PARA++ provides a generic high-level interface for performing message passing in C++, with no attempt (by design) to be consistent with MPI [1].

Overview. In this paper, we present Object-Oriented MPI (OOMPI), a C++ class library for MPI. The complete documentation for OOMPI can be found in [10]. Section 2 discusses the functional and non-functional requirements that

guided the design process for OOMPI. A detailed analysis of the goals and desired functionality is discussed in Section 3. Section 5 provides a simple example program to demonstrate the intuitive OOMPI interface.

2. Requirements

With the specification of a C++ class library [2, 6], we will necessarily be moving away from the simple one-to-one mapping of MPI function to language binding (as with C and Fortran). We therefore also run the risk of adding, losing, or changing MPI-1 specified functionality with the library specification.

In order to properly delimit the scope of the MPI C++ class library, we have the following guidelines:

Semantics. The MPI C++ class library must provide a semantically correct interface to MPI.

Syntax. The names of member functions should be consistent with the underlying MPI functions that are invoked by each member function.

Functionality. The MPI C++ class library must provide all functionality defined by MPI-1. To the greatest extent possible, this functionality should be provided through member functions of objects, although some globally scoped functions may be permitted.

Objects. It is only natural to think of communicating objects in a message-passing C++ program. The MPI-1 specification, however, does not deal with objects. It only specifies how *data* may be communicated. Thus, we require that the MPI C++ class library similarly provide the capability for sending the data that is contained within objects.

Moreover, since the data contained within an object is essentially a user-defined structure, we require that mechanisms be provided to build MPI-1 user-defined data types for object data and for communicating that data in a manner identical to communicating primitive data types. Objects that have complex data to be communicated must be explicitly constructed to do so.

Implementation. The MPI C++ class library must be a layer on top of the C bindings and C++ bindings (which are currently being finalized). In conjunction with the guidelines for functionality, this implies that the MPI-1 functionality will essentially be provided with calls to C functions. That is, there will be no attempts for the C++ class library itself to provide any MPI-1 functionality apart from that provided by the C bindings.

Further implementation stipulations are that: The class library must introduce as little overhead as possible to the C bindings. Second, the class library may not make use of internal details of particular implementations of C bindings. Third, except where the C++ language offers a simpler interface, preserve similar function names from the C MPI bindings as well as necessary arguments.

3. Analysis

3.1. Syntax

A typical MPI function call (in C) is of the following form:

```
MPI_Comm comm;
int i, dest, tag;
. . .
MPI_Send(&i, 1, MPI_INT, dest,
         tag, comm);
```

Here, `i`, `1`, `MPI_INT`, and `tag` specify the content and type of the message to be sent, and `comm` and `dest` specify the destination of the message. A more natural syntax results from encapsulating the pieces of information that make up the message and the destination. That is, we could perhaps encapsulate `i`, `1`, `MPI_INT`, and `tag` as a message object and `comm` and `dest` as a destination (or source) object.

Before committing to any objects, let's examine the sort of expressive syntax that we would like for `OOMPI`. The function call above would be very naturally expressed as

```
int i;
Send(i);
```

But this is incomplete — we still require some sort of destination object. In fact, we would like an object that can serve as both a source and a destination of a message. In `OOMPI`, this object is called a *port*.

3.2. Ports and Communicators

Using an `OOMPI_Port`, we can send and receive objects with statements like:

```
int i, j;
OOMPI_Port Port;
. . .
Port.Send(i);
Port.Receive(j);
```

The `OOMPI_Port` object contains information about its MPI communicator and the rank of the process to whom the message is to be sent. Note, however, that although the expression `Port.Send(i)` is a very clear statement of what we want to do, there is no explicit construction of a message object. Rather, the message object is implicitly constructed (see 3.3 below).

Port objects are very closely related to communicator objects — a port is said to be a communicator's view of a process. Thus, a communicator contains a collection of ports, one for each participating process. `OOMPI` provides an abstract base class `OOMPI_Comm` to represent communicators. Derived classes provided by `OOMPI` include `OOMPI_Intra_comm`, `OOMPI_Inter_comm`, `OOMPI_Cart_comm`, and `OOMPI_Graph_comm`, corresponding to an intracommunicator, intercommunicator, intracommunicator with Cartesian topology, and intracommunicator with graph topology, respectively.

Individual ports within a communicator are accessed with `operator[]()`, i.e., the `i`th port of an `OOMPI_Comm c` is `c[i]`. The following code fragment shows an example of sending and receiving:

```
int i, j, m, n;
OOMPI_Intra_comm Comm;
. . .
Comm[m].Send(i);
Comm[n].Receive(j);
```

Here, the integer `i` is sent to port `m` in the communicator and the integer `j` is received from port `n`.

3.3. Messages

We define an `OOMPI_Message` object with a set of constructors, one for each of the MPI base data types. Then, we define all of the communication operations in terms of `OOMPI_Message` objects. The need to construct `OOMPI_Message` objects explicitly is obviated — since promotions for each of the base data types are declared, an `OOMPI_Message` object will be constructed automatically (and transparently) whenever a communication function is called with one of the base data types.

The base types supported by `OOMPI` are:

```
char      short  unsigned char
long      int    unsigned short
unsigned  float  unsigned long
double
```

In addition to messages composed of single elements of these types, it is also desirable to send messages composed of arrays of these types. By introducing an `OOMPI_Array_message` object, we can also provide automatic promotion of arrays. Thus, to send an array of integers, we can use a statement like:

```
int a[10];
OOMPI_Port Port;
. . .
Port.Send(a, 10);
```

Again, no explicit message is constructed.

Note that in the above examples we have not explicitly given a tag to the messages that are sent. If no tag is given, a default tag is assigned by `OOMPI`, but a user can supply a tag as well:

```
int a[10];
OOMPI_Port Port;
. . .
Port.Send(a, 10, 201);
```

3.4. User Defined Data Types

Although it is convenient to be able to pass messages of arrays or of single elements of basic data types, significantly more expressive power is available by accommodating user objects (i.e., user-defined data types). That is, `OOMPI` should provide the ability to make statements of the form:

```
MyClass a[10];
OOMPI_Port Port;
. . .
Port.Send(a, 10, 201);
```

To accomplish this, `OOMPI` provides the base class `OOMPI_User_type` from which all non-base type objects that will be communicated must be derived. This class provides an interface to the `OOMPI_Message` and `OOMPI_Array_message` classes so that objects derived from `OOMPI_User_type` can be sent using the syntax above.

Besides inheriting from `OOMPI_User_type`, the user must also construct objects in the derived class so that an underlying `MPI_Datatype` can be built. `OOMPI` provides a streams-based interface to make this process easier. The following is an example of a user-defined `class` object:

```
class foo : public OOMPI_User_type {
public:
  foo() : OOMPI_User_type(type, this,
          FOO_TAG) {

    // Build the data type if it is not
    // built already
    if (!type.Built()) {
      type.Struct_start();
      type << a << b << c << d;
      type.Struct_end();
    }
  }
private:

  // The data for this class
  int a, b;
```

```
  double c, d;

  // Static variable to hold the newly
  // constructed MPI_Datatype
  static OOMPI_Datatype type;
};
```

The steps in making a user object suitable for use in `OOMPI` as an `OOMPI_Message` are:

1. Derive the class from `OOMPI_User_type`.
2. The object must contain a static `OOMPI_Datatype` member.
3. The constructor for the class must initialize `OOMPI_User_type` according to

   ```
   OOMPI_User_type(OOMPI_Datatype&
     type, USER_TYPE *this, int tag)
   ```

   where `type` is the name of the static `OOMPI_Datatype` member, and `tag` will be the default tag for all instances of this class.
4. Identify the internal data to be communicated.
5. The constructor for the class must check if the static `OOMPI_Datatype` member has been built (by calling its `Built()` member function). If it has not been built, appropriate calls to the datatype must be made so that it can be built.

Note that the tag that is set with the `OOMPI_User_type` constructor will apply (by default) all instances of the `foo` class. This default tag may be overridden.

3.5. Return Values and Error Handling

All functions in MPI-1 return an error condition; a provision for installing error handlers allows errors to be trapped in various specified ways. Again, C++ allows OOMPI to be more expressive. Rather than returning error codes, OOMPI functions have specified return values (typically corresponding to MPI-1 "out" parameters). Error conditions may then optionally be handled with exceptions.

OOMPI allows one of three actions to happen upon an MPI error: the underlying MPI implementation may handle the error, OOMPI may throw an exception, or OOMPI may simply set `OOMPI_errno` and return. These three functions can be set per communicator; see the `OOMPI_Comm` class for more details.

If the function returns after the error has been handled, OOMPI will attempt to return an invalid value depending on the type of object being returned. For example, functions that return pointers or arrays will return `0` (casted to the appropriate type). Functions returning `int` will return `OOMPI_UNDEFINED`. Functions that return OOMPI objects will return invalid objects; attempting to invoke any member functions on them will result in another error.

3.6. A Stream Interface for Message Passing

Streams are a standard mechanism used in C++ for performing I/O. The syntax of stream based I/O is appropriate for message passing. That is, messages can be sent and received in `OOMPI` with the statements:

```
int i, j;
OOMPI_Port Port;
. . .
Port << i << j;
Port >> i >> j;
```

Since second arguments to `operator>>()` and `operator<<()` are not possible, default tags (based upon the message type) are used. These default tags can be overridden, however. See section 4.2 for the discussion of the `OOMPI_Message` and `OOMPI_Array_message` objects. Note that user-defined data types can have their default tags set by the user with the `Set_tag()` member function. To enforce thread safety, each variable is sent or received individually using the `MPI_Send()` or `MPI_Recv()` function calls. In the above example, `i` is sent with `MPI_Send()`, `j` is sent with `MPI_Send()`, `i` is received with `MPI_Recv()`, and finally `j` is received with `MPI_Recv()`.

3.7. Packed Data

MPI-1 provides the capability for users to pack their own messages. A stream interface is provided in `OOMPI`. In the following example, a message of 200 integers is constructed and sent:

```
int i;
OOMPI_Port Port;
OOMPI_Packed msg(
  OOMPI_COMM_WORLD.Pack_size(i, 200),
  OOMPI_COMM_WORLD, PACK_TAG);
. . .
msg.Start();
for (i = 0; i < 200; i++)
  msg << i << rank;
msg.End();
Port << msg;
```

The arguments to the `OOMPI_Packed` constructor are the size of the buffer to be created, the communicator, and the tag to be used for sending and receiving this instance. Note that no `count` argument is passed to the `Port` when sending the object; an `OOMPI_Packed` object inherently

knows its count. That is, sending an `OOMPI_Packed` object will send as many bytes as were packed. Receiving an `OOMPI_Packed` object will attempt to receive a message as long as the entire buffer. MPI-1 allows the normal receipt of a shorter-than-expected message.

Note that the `OOMPI_Packed` object has local state, which would make it non thread safe. However, it does not make sense for more than one thread to pack into the same buffer. Therefore, we define a process that has multiple threads packing into one buffer erroneous. Each thread should pack into its own `OOMPI_Packed` instance. In any case, the `OOMPI_Packed` object provides the same level of thread safety for packing as does MPI-1.

3.8. Attributes

OOMPI does not support MPI attributes. The MPI functions `MPI_Attr_Get()`, `MPI_Attr_Put()`, `MPI_Keyval_create()`, and `MPI_Keyval_free()`, have no corresponding functions or classes in OOMPI. Attribute caching can be handled in C++ in a much more efficient and intuitive manner than is provided with the MPI interface. Future versions of OOMPI may include some attribute caching scheme.

4. Objects

Listed below are all the objects that are mentioned briefly above. Each object contains a brief description and list of functional requirements.

Each object is prefixed with `OOMPI_` so that no name conflicts will occur with the ANSI C bindings of functions, datatypes, and constants. All OOMPI names (member functions, objects, and constants) follow the same capitalization scheme as MPI-1 names. In addition to the `MPI_` prefix, many MPI-1 functions also contained a second prefix to classify functionality (e.g., `MPI_Type_*`). In such cases, the second prefix was made part of the object name and the member functions were named from the remaining suffix. For example, `MPI_Type_Vector()` became the `Vector()` member function of the `OOMPI_Datatype` object.

4.1. Communicator Objects

The objects associated with communicators are:

`OOMPI_Comm`	`OOMPI_Comm_world`
`OOMPI_Group`	`OOMPI_Intra_comm`
`OOMPI_Cart_comm`	`OOMPI_Port`
`OOMPI_Inter_comm`	`OOMPI_Graph_comm`
`OOMPI_Any_port`	

These objects encapsulate the functionality of MPI communicators and are the basis for all communication (point-to-point and collective). The communicator objects contain the algebraic group object used to create the communicator, a port object for each rank in the communicator, and an error handler (if there is one).

The `OOMPI_Comm` object is an abstract base class from which the `OOMPI_Intra_comm`, `OOMPI_Inter_comm`, and `OOMPI_Comm_world` classes are derived. These classes represent and provide the functionality associated with intra-communicators, inter-communicators, and `MPI_Comm_world`, respectively. Note that the class `OOMPI_Comm_world` has only one instance of an object, the global variable `OOMPI_COMM_WORLD`.

The `OOMPI_Group` object encapsulates all the operations on groups. A group in MPI is an ordered set of process identifiers. In OOMPI, the `OOMPI_Group` is used by the `OOMPI_Communicator` object.

An `OOMPI_Port` object is created for each rank in a communicator. It encapsulates all the point-to-point and rooted collective communication functionality. Point-to-point communication routines (e.g., `Send()` and `Recv()`) invoked on an `OOMPI_Port` implicitly specify the destination (or source) rank. Rooted collective communication routines invoked on an `OOMPI_Port` implicitly specify the root of the operation.

4.2. Message and Data Objects

The `OOMPI` objects associated with messages are:

`OOMPI_Message`	`OOMPI_User_type`
`OOMPI_Request`	`OOMPI_Array_message`
`OOMPI_Packed`	`OOMPI_Status`
`OOMPI_Datatype`	`OOMPI_Op`

The MPI-1 C bindings of MPI-1 specify that all data buffers are of type `(void *)`. Since the type of the data is not inherent in the argument, a second argument must be specified to provide the type. In C++, functions can be overloaded based on the type of their formal parameters, but there are two problems with this approach: it leads to a function explosion and user-defined types are not included in this scheme. Using `OOMPI_Message` as a base class with lightweight default promotions for all base types provides a clean, efficient, and useful way to not have to overload functions for each type.

The `OOMPI_Message` object is a base class that is used to unify diverse data types (base C++ types and user-defined types) into one object type. That is, every MPI-1 function that includes a `(void *)` data buffer argument is replaced with an `OOMPI_Message` argument (and/or `OOMPI_Array_message` argument, see below). Since the `OOMPI_Message` object includes the MPI datatype and a pointer to the top of the data, functions that have

`OOMPI_Message` arguments inherently know the data's type and where it resides in memory.

The `OOMPI_Message` object can be used for both implicit promotion and explicit message formation. It is sometimes desirable to explicitly form an `OOMPI_Message` to override a default type tag or to encapsulate an entire array (to include the `count` argument). The resulting `OOMPI_Message` object can be re-used after it is formed, even if the variable (or values in the array) change; the `OOMPI_Message` object keeps a pointer to the data just for this purpose.

The `OOMPI_Array_message` object is very similar to the `OOMPI_Message` object except that it is used to *implicitly* promote arrays. It does *not* take an argument indicating how many elements exist in the array; `OOMPI_Array_message` is only used as a *promotion* mechanism, and can therefore only take one argument.

One of the main reasons for splitting the implicit promotion of arrays into its own class is to avoid an ambiguity where count arguments are required. Since the `OOMPI_Array_message` class is *only* used for promotion purposes, an explicit `count` argument must be supplied. The OOMPI equivalents of the `MPI_Send()` function are declared below. In the second function, the `count` argument specifies how many elements are in the array.

```
void Send(OOMPI_Message buf,
          int tag = OOMPI_NO_TAG);
void Send(OOMPI_Message_array buf,
          int count,
          int tag = OOMPI_NO_TAG);
```

`OOMPI_Array_message` is *only* used as an internal object; it is not considered to be part of the user interface.

That is, there are three mechanisms to pass data of base C++ types (user defined types are discussed in section 3.4) to OOMPI functions: two implicit mechanisms and one explicit mechanism. `OOMPI_Message` and `OOMPI_Array_message` are used to implicitly promote the base types. Note that the implicit promotion to an `OOMPI_Array_message` is not sufficient for the stream interface because the `count` argument cannot be supplied.

```
int i, j[10];
OOMPI_Port Port;
. . .

// Both implicit mechanisms can be
// used with the standard interface:
Port.Send(i);
Port.Send(j, 10);

// Only the implicit scalar
// promotion can be used with
// the stream interface:
Port << i;
```

`OOMPI_Message` can also be used to explicitly create re-usable messages that contain either scalar variables or arrays:

```
int i, j[10];
OOMPI_Port Port;
OOMPI_Message imsg(i, MY_INT_TAG);
OOMPI_Message jmsg(j, 10, MY_TAG);

// Explicitly formed messages can
// be sent through the standard
// interface:
Port.Send(imsg);

// Or they can be sent through the
// stream interface:
Port << jmsg;

// They can also be re-used:
i++; j[3]++;
Port << imsg << jmsg;
```

The `OOMPI_Datatype` object is used to describe the datatype of a message. In addition to providing access to functions that build the less complicated user-defined datatypes such as `MPI_Type_contiguous()` and `MPI_Type_vector()`, the `OOMPI_Datatype` object also provides a simple, streams-based interface to build more complex datatypes with `MPI_Type_struct()`. The `OOMPI_Datatype` object can build and commit any valid user-defined MPI-1 datatype.

The `OOMPI_User_type` object is the heart of user-defined datatypes. It must be inherited and initialized by all objects that will be sent and/or received in message passing calls (refer to Section 3.4 . It is very similar to `OOMPI_Message` in that it is used to unify all datatypes (through inheritance) into a single type that can be used to access the object's type and data.

The `OOMPI_Packed` object provides a simple, streams-based interface for packing and unpacking messages (see Section 3.7. The buffer that is used for packing and unpacking can either be specified by the user or allocated by the `OOMPI_Packed` object.

The `OOMPI_Op` object is a simple wrapper to the `MPI_Op_create()` and `MPI_Op_free()` functions.

`OOMPI_Request` objects are used for non-blocking communications to identify a posted communication and match the initiating post with the post that terminates it. A request object identifies properties of a communication operation such as send mode, the communication buffer, its context, and the tag and destination arguments to be used for a send (or receive). In addition, this object stores in-

formation about the status of the pending communication operation.

The `OOMPI_Status` object encapsulates all the operations that can be performed on an `MPI_Status` handle. These operations include the MPI functions `MPI_Get_count()`, `MPI_Get_elements()`, and functions for determining the source and type of incoming messages.

4.3. Object Semantics

The semantics of OOMPI object is a critical issue. All of the objects exist to provide access to MPI-1 functionality, so the semantics of their member functions are well defined. However, it is not completely clear what happens in the presence of some of the expressive power that we gain by using C++.

For instance, it is important to define what happens in the following sort of statement:

```
int i;
OOMPI_Intra_comm a;
a = OOMPI_COMM_WORLD;
a[0].Receive(i);
```

In particular, here are some questions about the above statement:

1. In the statement `OOMPI_Intra_comm a`, what value is given to the internal MPI communicator handle of `a`?
2. What would happen if a communication operation were attempted using `a` just following its construction?
3. In the statement `a = OOMPI_COMM_WORLD`, what value is given to the internal communicator of `a`? Is it `MPI_COMM_WORLD` or is it a duplicate (using `MPI_Comm_dup()`)? What happens to the internal communicator that might already exist in `a`? What if another object references that communicator?

These and other issues are handled by a set of formalisms for construction, destruction, copying, and assignment of `OOMPI` objects.

Handles. Most `OOMPI` objects encapsulate MPI handles and their associated functions. As such, it is very important to provide sharing semantics for the underlying MPI handles. For example, consider a statement like

```
int i = 0;
OOMPI_Request r;
r = OOMPI_COMM_WORLD[0].Send_init(i);
```

The call to the `Send_Init()` member function of `OOMPI_Port` ultimately results in a call to `MPI_Send_init()`. The call to `MPI_Send_init()` will in turn produce an `MPI_Request` handle that is then wrapped up inside an `OOMPI_Request` which is the return value of `Send_init()`. This return value is then assigned to `Request`. Since the underlying `MPI_Request` is an opaque handle, it is very important that `Request` contain the same internal `MPI_Request` handle as the object returned by `Send_init()`.

`OOMPI` includes a simple internal reference counting mechanism for providing such sharing semantics. The internal MPI handles of `OOMPI` objects are not themselves contained inside of `OOMPI` objects (although it is useful to consider them to be). Rather, they are wrapped up in a special container object and the `OOMPI` objects themselves have a "smart pointer" to the wrapped-up handle to effect reference counting.

The ramifications of the sharing semantics on the construction, destruction, copying, and assignment of `OOMPI` objects are described below.

Construction and Destruction. All `OOMPI` objects that have internal MPI handles provide a constructor that takes the corresponding MPI handle as an argument. However, this constructor should only be used for compatibility with existing MPI libraries (see below). Most OOMPI objects are referenced counted, where each new reference to the internal MPI handle causes the reference count to be incremented. When a reference counted `OOMPI` object is deleted or goes out of scope, the reference count of the container is decremented. The object is freed when the count reaches zero.

Compatibility. In order to maintain compatibility with existing MPI C libraries, it is not only necessary to be able to construct OOMPI objects from MPI handles (as discussed above), it may also be necessary to extract the MPI-1 handle from the OOMPI object. For such cases, any OOMPI object that contains an MPI handle also includes a `Get_mpi(void)` member function which will return a reference to the internal MPI handle. It should be noted that the `Get_mpi()` function is *only* intended to provide an interface to the underlying MPI objects for use by external libraries. Extracting the underlying MPI object and using it for the construction of another OOMPI object will create inconsistencies within OOMPI because of the reference counting mechanism.

`const` **Semantics.** This version of OOMPI was specifically designed to be implemented on top of existing MPI-1 ANSI C bindings. As such, it was impossible to use `const` for functions and arguments in OOMPI when the underlying MPI implementation did not make use of it at all. Since MPI-2 will include C++ bindings which will most likely make use of `const`, future versions of OOMPI can be layered on the C++ bindings rather than the C bindings, and therefore take advantage of `const` semantics.

5. Example Program

We demonstrate the use of OOMPI with the ever-popular ring program. The root process introduces a message which travels around the ring-ordered processes a given number of times before stopping. This program shows two features from OOMPI: ports and stream based communication.

```
#include "oompi.h"
int main(int argc, char *argv[])
{
  int count = 5, msg = 123;
  OOMPI_COMM_WORLD.Init(argc, argv);

  int rank = OOMPI_COMM_WORLD.Rank();
  int size = OOMPI_COMM_WORLD.Size();
  int to   = (rank + 1) % size;
  int from = (size + rank - 1) % size;

  OOMPI_Port pto, pfrom;
  pto = OOMPI_COMM_WORLD[to],
  pfrom = OOMPI_COMM_WORLD[from];

  // Process 0 is sent the first message
  if (rank == size - 1)
    pto << msg;

  for (int i = 0; i < count; i++) {
    // the process receives the message
    pfrom >> msg;
    // the process sends the message
    pto << msg;
  }

  // Process 0 receives the last message
  if (rank == 0)
    pto << msg;

  OOMPI_COMM_WORLD.Finalize();
  return 0;
}
```

6. Availability

Information about releases, documentation, and patches is maintained at: `http://www.cse.nd.edu/~/lsc/research/oompi/`

7. Acknowledgments

The authors would like to acknowledge many helpful discussions with Tony Skjellum and other members of the MPI Forum. This work was supported by NSF grants ASC94-22380 and CCR95-02710.

References

[1] Olivier Coulaud and Eric Dillon. PARA++: C++ Binding for Message Passing Libraries User Guide. Technical report, INRIA, 1995.

[2] C++ Forum. Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++. Technical report, American National Standards Institute, 1995.

[3] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[5] Dennis Kafura and Liya Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI Developers Conference*, Notre Dame, IN, June 1995. `http://www.cse.nd.edu/mpidc95/-proceedings/papers/html/huang/`.

[6] The Annotated C++ Reference Manual. *Margaret A. Ellis and Bjarne Stroustrup*. Addison Wesley, 1990.

[7] Message Passing Interface Forum. MPI-2, July 1995. `http://www.mcs.anl.gov/Projects/-mpi/mpi2/mpi2.html`.

[8] Anthony Skjellum, Ziyang Lu, Purushotham V. Bangalore, and Nathan E. Doss. Explicit parallel programming in C++ based on the message-passing interface (MPI). In Gregory V. Wilson, editor, *Parallel Programming Using C++*. MIT Press, 1996. in press. Also available as MSSU-EIRS-ERC-95-7.

[9] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MPI, 1996.

[10] Jeffrey M. Squyres, Brian C. McCandless, and Andrew Lumsdaine. Object Oriented MPI Reference. Technical Report TR 96-10, Department of Computer Science and Engineering at the University of Notre Dame, 1996.