Object Oriented MPI (OOMPI):
A C++ Class Library for MPI

Jeffrey M. Squyres
Jeremiah Willcock
Brian C. McCandless
Peter W. Rijks
Andrew Lumsdaine
http://www.osl.iu.edu/research/oompi/

Open Systems Laboratory
Pervasive Technologies Labs
Indiana University

September 3, 2003
## Contents

1 Overview ................................. 4
2 Requirements ............................ 4
3 Analysis ................................. 5
   3.1 Syntax .................................. 5
   3.2 Ports and Communicators .......... 5
   3.3 Messages ............................... 6
   3.4 User Defined Data Types .......... 7
   3.5 Return Values ......................... 9
   3.6 A Stream Interface for Message Passing 9
   3.7 Packed Data ......................... 10
   3.8 Attributes ............................. 10
   3.9 Objects ................................. 10
   3.10 Communicator Objects ............. 11
   3.11 Message and Data Objects ....... 11
   3.12 Object Semantics ................. 13
4 Examples ................................. 15
   4.1 Ring, Version 1 ...................... 15
   4.2 Ring, Version 2 ...................... 16
5 Design .................................. 17
   5.1 Notation ............................... 18
   OOMPI_Array_message .................. 18
   OOMPI_Cart_comm ....................... 21
   OOMPI_Comm .............................. 24
   OOMPI_Comm_world ..................... 26
   OOMPI_Datatype .......................... 28
   OOMPI_Environment ..................... 32
   OOMPI_Error ............................. 33
   OOMPI_Graph_comm ...................... 34
   OOMPI_Group ............................. 36
   OOMPI_Inter_comm ...................... 39
   OOMPI_Intra_comm ...................... 40
   OOMPI_Message ........................... 43
   OOMPI_Op ................................. 45
   OOMPI_Packed ............................ 47
   OOMPI_Port ............................... 49
   OOMPI_Request ........................... 53
   OOMPI_Request_array ................... 55
   OOMPI_Status ............................. 58
   OOMPI_Status_array .................... 60
   OOMPI_Tag ................................. 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOMPI_User_type</td>
<td>62</td>
</tr>
<tr>
<td>OOMPI Enumerated Types</td>
<td>63</td>
</tr>
<tr>
<td>OOMPI Constants</td>
<td>65</td>
</tr>
<tr>
<td>6 Availability</td>
<td>67</td>
</tr>
<tr>
<td>6.1 The OOMPI Distribution</td>
<td>67</td>
</tr>
<tr>
<td>6.2 Contact Information</td>
<td>67</td>
</tr>
<tr>
<td>Index</td>
<td>68</td>
</tr>
<tr>
<td>References</td>
<td>68</td>
</tr>
</tbody>
</table>
1 Overview

This paper describes an object oriented approach to the Message Passing Interface (MPI) [2–6]. Object Oriented MPI (OOMPI) is a class library specification that encapsulates the functionality of MPI into a functional class hierarchy to provide a simple, flexible, and intuitive interface.

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 for an MPI class library is discussed in Section 3. Section 4 provides two examples of OOMPI programs that demonstrate its intuitive interface, while Section 5 provides manual pages for all the classes and member functions in OOMPI. Finally, Section 6 announces the release of an open-source implementation of OOMPI, and provides contact information for obtaining the distribution package and contacting the authors.

2 Requirements

With the specification of a C++ class library [1,7], 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\textsuperscript{1}. 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.

\textsuperscript{1}For consistency, this document only discusses OOMPI relative to the C MPI bindings. Although C++ MPI bindings are now available, OOMPI has not yet been implemented on top of them. Future plans for OOMPI include implementations based upon the MPI C++ bindings – additional OOMPI documentation will be released at that time.
Further implementation stipulations are that:

- The class library must introduce as little overhead as possible to the C bindings.
- The class library may not make use of internal details of particular implementations of C bindings.
- 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:

```c
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

```c
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:

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

---

2The moniker “port” was suggested by Marc Snir.
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 intra-communicator, inter-communicator, intra-communicator with Cartesian topology, and intra-communicator with graph topology, respectively.

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

```c
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.

**Discussion:** Message objects could be eliminated entirely by declaring each communication function in terms of every base data type. However, this would result in an enormous number of almost identical member functions. The use of message objects seems better for the sake of maintainability. There is some function overhead because of the need to construct a message object, but the constructors can be made very lightweight so that the overhead is negligible.

The base types supported by OOMPI are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>short</td>
<td>int</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td>unsigned short</td>
<td>unsigned</td>
<td>unsigned long</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>double</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```c
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:

```cpp
int a[10], tag;
OMPI_Port Port;
...

Port.Send(a, 10, tag);
```

The declaration of `OMPI_Port::Send()` is

```cpp
void OOMPI_Port::Send(OOMPI_Message buf, int tag = OOMPI_NO_TAG);
void OOMPI_Port::Send(OOMPI_Array_message buf, int count,
int tag = OOMPI_NO_TAG);
```

Here, the default value of `OMPI_NO_TAG` is not the tag used on the message. Rather, it is a sentinel
value that indicates that no tag was explicitly given, so inside the body of `OMPI_Send()`, a default tag is
used, depending on the type of the data. OOMPI reserves the top `OMPI_RESERVED_TAGS` tags. Users
can use any tag between zero and `OMPI_TAG_UB`.

### 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:

```cpp
MyClass a[10], tag;
OMPI_Port Port;
...

Port.Send(a, 10, tag);
```

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

Besides inheriting from `OMPI_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:

```cpp
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(this);
    type << a << b;
    type << c << d << e;
    type.Struct_end();
}

private:

    // The data for this class
    int a, b;
    double c, d;
    char e;

    // 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, 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 with the Set_tag(int tag) member function for particular instances of foo.3

Discussion: To make the building of datatypes thread safe, the entire process must be protected by a mutex. The Built() member function performs a down on the mutex before checking to see if the datatype has been built or not. If it has not been built yet, Built() returns a FALSE and the type is built. When Struct_end() is invoked,

3The functions Get_tag() and Set_tag() are inherited from the OOMPI_Tag class, which is described in Section 5.1.
MPI_Type_Struct() and MPI_Type_commit() are called to build the datatype, and the up is performed on the mutex.

3.5 Return Values

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:

```cpp
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 3.11 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().

Discussion: The streams interface could be expanded to include many more features (such as tags to indicate the end of a message, tags to indicate what type of send/receive should be used, etc.). However, none of these concepts are thread safe since they imply that the OOMPI_Port object must contain local state. On the other hand, the standard could specify that threads must keep their own copies of ports, but this would break the similarity between ports and send operations.
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 400 integers is constructed and sent:

```c
int i, rank = OOMPI_COMM_WORLD.Rank();
OOMI_Port Port;
OOMI_Packed msg(OOMPI_COMM_WORLD.Pack_size(i, 400),
                 OOMPI_COMM_WORLD, PACK_TAG);
... 
```

```c
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.

Discussion: Note that the OOMPI_Packed object has local state. 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.

3.9 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.
3.10 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 classes OOMPI_Intra_comm, OOMPI_Inter_comm, and OOMPI_Comm_world 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.

Discussion: For MPI-2, it might be desirable to create a one-way communicator class that is also derived from the OOMPI_Comm object.

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.

3.11 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).
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 encapsu-
late an entire array (to include the count argument). The resulting OOMPI_Message object can be re-used
after it is formed, even if the value of the variable (or values in the array) changes; 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.

```c
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.

```c
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. This reduces the amount overheard generated due to promotions; explicitly formed
messages should be used when the same memory will be used to send or receive messages multiple times.

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

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

// Or they can be sent through the stream interface:
Port << imsg;
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. 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. 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 information 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.

3.12 Object Semantics

The semantics of OOMPI objects 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

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

The call to the `Send_Init()` member function of `OOMPI_COMM_WORLD` 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.** All OOMPI objects that have internal MPI handles will provide a constructor that takes the corresponding MPI handle as an argument. The argument will have a default value of the handle NULL value. For example, this constructor for `OOMPI_Cart_comm` would be declared as:

```c
OOMPI_Cart_comm(MPI_Comm mpi_comm = MPI_COMM_NULL);
```

**Destruction.** Destruction of an OOMPI object with a smart pointer (and concomitant destruction of the smart pointer itself) will cause the reference count of the container to be decremented. A decrement to zero will cause the container itself to be destroyed and a pre-defined function to be called on the handle contained therein (i.e., the corresponding `MPI_*Free()` function will be called for that handle, for example `MPI_Request_free()`).
Copying and Assignment. A copy or an assignment is usually two steps; 1) destruction of the previous contents, 2) assignment of the new contents. Step 1 is discussed in the previous paragraph; step 2 is simply the inverse — increment the reference count of the container that is being copied.

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.

Discussion: 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 inconsistency problems within OOMPI. Since OOMPI uses a wrapping scheme to ensure that separate instances of OOMPI objects actually point to the same MPI object, using the extracted MPI object to create another OOMPI object will create second wrapper instance within OOMPI rather than a copy of the original wrapper. This is considered erroneous.

Care should also be taken that the original OOMPI object exists for the entire time that the handle obtained from Get_mpi() is used; when the OOMPI object is destroyed, it may invoke the corresponding MPI_*free() function (as described above). This would render the handle obtained from Get_mpi() invalid.

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 certainly make use of const, future versions of OOMPI can be layered on the C++ bindings rather than the C bindings, and therefore utilize const constructs.

inline. This document only outlines the design requirements for OOMPI; it does not specify particular implementation details. As such, inline is an optimization that will be expected in high-quality OOMPI implementations, but it is not required, and therefore is not specified in this document. Since OOMPI is a thin layer on top of existing MPI bindings, it only makes sense to use inline wherever possible, but this is an implementation decision.

4 Examples

Here we present some small examples of parallel programs written using OOMPI.

4.1 Ring, Version 1

The first example is of the ubiquitous ring program.
#include <iostream>
#include "oompi.h"

using namespace std;
int
main(int argc, char *argv[])
{
    int count = 5;
    OOMPI_COMM_WORLD.Init(argc, argv);
    int i = 0;
    int msg = 123;

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

    cout << "I am node " << rank << " of " << size << endl;
    cout << "Sending to " << to << " and receiving from " << from << endl;

    if (rank == size - 1)
        OOMPI_COMM_WORLD[to].Send(msg);

    for (i = 0; i < count; i++) {
        OOMPI_COMM_WORLD[from].Recv(msg);
        cout << "Node " << rank << " received " << msg << endl;
        OOMPI_COMM_WORLD[to].Send(msg);
    }

    if (rank == 0) {
        OOMPI_COMM_WORLD[from].Recv(msg);
        cout << "Node " << rank << " received " << msg << endl;
    }

    cout << "All done!" << endl;
    OOMPI_COMM_WORLD.Finalize();

    return 0;
}

4.2 Ring, Version 2
The following program is essentially the same as the previous one, but some more features of OOMPI are used. Ports are created and used for the stream-based communication.
#include <iostream>
#include "oompi.h"

using namespace std;

int
main(int argc, char *argv[]) {
    int count = 5;
    int msg = 123;
    OOMPI_COMM_WORLD.Init(argc, argv);

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

    cout << "I am node " << rank << " of " << size << endl;
    cout << "Sending to " << to.Rank() << " and receiving from " << from.Rank() << endl;

    if (rank == size - 1)
        to << msg;
    for (int i = 0; i < count; i++) {
        from >> msg;
        cout << "Node " << rank << " received " << msg << endl;
        to << msg;
    }

    if (rank == 0) {
        from >> msg;
        cout << "Node " << rank << " received " << msg << endl;
    }

    cout << "All done!" << endl;
    OOMPI_COMM_WORLD.Finalize();

    return 0;
}

5 Design

In this section, the functional interface to each OOMPI object will be described in detail; all public member functions will describe what they do, what MPI-1 functions are used to implement it, and what kinds of copies are performed (if appropriate). An index is provided at the end of this annex to cross reference MPI-1 functions to their corresponding OOMPI objects.

Since most MPI-1 objects (e.g., MPI_Comm) are simply handles to opaque data, they cannot be directly copied. Therefore, any OOMPI object that contains an MPI-1 handle has two choices when making a copy of itself: invoke the appropriate MPI-1 function to copy the opaque data, or use a reference counting scheme that will provide references to the handle (finally invoking the appropriate MPI_* free() function when
all references have been deleted). The different approaches are known as “deep” and “shallow” copies, respectively. With the exception of the OOMPI_Packed, OOMPI_Request, OOMPI_Request_array, OOMPI_Status, and OOMPI_Status_array classes, all copy constructors and assignment operators of OOMPI objects with an internal MPI-1 handle perform shallow (reference counted) copies.

See Section 6 for a information about V ersion 1.0.4 of an open-source implementation of OOMPI.

5.1 Notation

Many functions in MPI take a choice argument, or pointer to a buffer. OOMPI uses the OOMPI_Message promotion mechanism to effect choice buffers for all scalar arguments, explicitly declared formally declared arrays, and (OOMPI_Array_message, count) pairs for arrays. As such, choice buffers may be expressed with one or two arguments. Since C++ allows the use of function overloading, OOMPI typically has multiple versions of functions to allow for this behavior. For example, the Send() function has two prototypes:

void Send(OOMPI_Message msg, int tag);
void Send(OOMPI_Array_message msg, int count, int tag);

However, the situation gets more complicated for functions that have two choice arguments:

OOMPI_Status Sendrecv(OOMPI_Message sendbuf, int dest, int sendtag, OOMPI_Message recvbuf, int source, int recvtag);
OOMPI_Status Sendrecv(OOMPI_Array_message sendbuf, int sendcount, int dest, int sendtag, OOMPI_Message recvbuf, int source, int recvtag);
OOMPI_Status Sendrecv(OOMPI_Message sendbuf, int dest, int sendtag, OOMPI_Array_Message recvbuf, int recvcount, int source, int recvtag);
OOMPI_Status Sendrecv(OOMPI_Array_message sendbuf, int sendcount, int dest, int sendtag, OOMPI_Message recvbuf, int recvcount, int source, int recvtag);

For the sake of clarity in the following pages, since OOMPI typically has overloaded functions for all possible combinations of the choice buffer types, choice buffer arguments will be denoted by the SMALL-CAPS font, with the understanding that any choice argument can be either an (OOMPI_Message msg) or an (OOMPI_Array_message msg, int count) pair:

void Send(SENDBUF, int tag);
void Sendrecv(SENDBUF, int dest, int sendtag, RECVBUF, int source, int recvtag);
Name  OOMPI_Array_message

Declaration  
#include "oompi.h"

class OOMPI_Array_message

Inheritance  None.

Description  The OOMPI_Array_message class is used for promotion of pointers only. This class is not considered to be part of the user interface; it should never be explicitly instantiated.

Constructors/Destructors

OOMPI_Array_message(<TYPE> data[]). Constructor. Extract the type and location of the argument. Since this constructor is only meant for promotion purposes, it cannot extract count information. This function is not templated; the <TYPE> notation is only used for brevity. <TYPE> can be any of the following base types:

<table>
<thead>
<tr>
<th>char</th>
<th>unsigned char</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>unsigned short</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>unsigned long</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A default tag, based upon the datatype, is used.
Access and Information

MPI_Datatype Get_type(void). Returns the MPI_Datatype of the argument to the constructor. Only meant to be used within OOMPI; this function is not part of the user interface.

void *Get_top(void). Returns a pointer of the argument to the constructor. Only meant to be used within OOMPI; this function is not part of the user interface.

See Also OOMPI Constants, OOMPI_Message
Name: OOMPI_Cart_comm

Declaration:
```c
#include "oompi.h"
class OOMPI_Cart_comm
```

Description:
A class for MPI communicators with Cartesian topology.

Inheritance:
This class is derived from OOMPI_Intra_comm.

Constructors/Destructors:

OOMPI_Cart_comm(MPI_Comm comm = MPI_COMM_NULL). MPI constructor. If comm is a valid communicator with an associated Cartesian topology, the constructor for the OOMPI_Comm base class is invoked with comm. Otherwise, it is invoked with MPI_COMM_NULL and the created OOMPI_Cart_comm is equivalent to OOMPI_COMM_NULL.

OOMPI_Cart_comm(const OOMPI_Cart_comm& a). Copy constructor. The copy constructor for the OOMPI_Comm base class is invoked.

OOMPI_Cart_comm& operator=(const OOMPI_Cart_comm& a). Assignment operator. The assignment operator for the OOMPI_Comm base class is invoked.

OOMPI_Cart_comm(OOMPI_Intra_comm& intra_comm, int ndims, int dims[], bool periods[], bool reorder = false). Constructor. Uses MPI_Cart_create() to construct an MPI communicator having a Cartesian topology from the intra-communicator intra_comm. A new container is created for the new communicator and a reference to it is made from this object.

OOMPI_Cart_comm(OOMPI_Intra_comm& intra_comm, int ndims, int dims[], bool periods, bool reorder = false). Constructor. Same as the constructor above, except that it uses a scalar periods argument to indicate whether all dimensions are periodic or not.

~OOMPI_Cart_comm(). Destructor. Destruction is handled by the OOMPI_Comm base class.

Communicator Management:

OOMPI_Cart_comm Dup(void). Returns a new OOMPI_Cart_comm that was created with the result of MPI_Comm_dup().

OOMPI_Cart_comm Sub(int remain_dims[]). Calls MPI_Cart_sub() and returns the new communicator.

OOMPI_Cart_comm Sub(bool dim0, ...). Similar to the function above, but expects ndims booleans instead of an array.
Access and Information

int Dim_get(). Calls MPI_Cartdim_get() and returns the number of dimensions in the current OOMPI_Cart_comm instance.

void Get(int maxdims, int dims[], bool periods[] = 0, int coords[] = 0). Calls MPI_Cart_get() and returns dimension array. periods and/or coords may not be specified if the return information is not necessary.

void Get(int dims[], bool periods[] = 0, int coords[] = 0). Shortcut to the previous function, except that the maxdims argument is not specified; ndims is used in its place.

int Rank(int coords[]). Calls MPI_Cart_rank() and returns the rank.

int Rank(int coord0, ...). Similar to the previous function, except that it expects a list of integer coordinates instead of an array. There must be ndims integers in the argument list.

int Rank(void). Returns the rank of the process by calling OOMPI_Intra_comm::Rank().

OOMPI_Port operator()(int coords[]). Calls MPI_Cart_rank() to get the rank of the neighbor specified by the coords array and returns the OOMPI_Port associated with it. coords must be ndims long.

OOMPI_Port operator()(int coord0, ...). Calls MPI_Cart_rank() to get the rank of the neighbor specified by (coord0, ...) and returns the OOMPI_Port associated with it. There must be ndims coordinates supplied in the argument list.

void Coords(int rank, int maxdims, int coords[]). Calls MPI_Cart_coords() and fills coords with the coordinates of rank.

void Coords(int rank, int coords[]). Shortcut to the previous function, except that ndims is used for maxdims.

void Coords(int coords[]). Shortcut to the previous function, except that the rank of the current process is used for rank.

int *Coords(int rank, int maxdims). Calls MPI_Cart_coords() and returns an array with the coordinates of rank.

int *Coords(int rank). Shortcut to the previous function, except that ndims is used for maxdims.

int *Coords(void). Shortcut to the previous function, except that the rank of the current process is used for rank.
int Shift(int direction, int disp, int& rank_source). Calls MPI_Cart_shift() and returns rank of destination and fills rank_source with the source.

int Shift(int direction, int disp). Shortcut to the previous function, but rank_source is not specified.

See Also OOMPI_Comm, OOMPI_Intra_comm, OOMPI_Graph_comm, OOMPI_Port
Name: OOMPI_Comm

Declaration:
#include "oompi.h"
class OOMPI_Comm

Description:
An abstract base class for MPI communicators. This class is responsible for maintaining an internal MPI communicator, information about the group associated with the communicator, and an array of OOMPI_Ports (one port for each MPI process having a rank in the communicator).

Inheritance:
None.

Constructors/Destructors:

OOMPI_Comm(MPI_Comm comm = MPI_COMM_NULL). MPI constructor. A new container is created for comm and a reference to it is made. This constructor also acts as the default constructor. The default error action OOMPI_ERRORS_ARE_FATAL is associated with the communicator.


OOMPI_Comm& operator=(const OOMPI_comm& a). Assignment operator. Perform a shallow (reference counted) assignment. The current reference to the internal MPI_Comm is deleted (which may trigger a call to MPI_Comm_free()).

virtual ~OOMPI_Comm(). Destructor. The destructor deletes all of its OOMPI_Ports and then deletes the reference to the container of its internal MPI communicator (which may result in a call to MPI_Comm_free()). If any copies of ports from this communicator still exist outside of the scope of this instance, they will remain valid. That is, the call to MPI_Comm_free() will be postponed until all referring communicators and ports have been deleted.

Access and Information:

MPI_Comm& Get_mpi(void). Returns a reference to the internal MPI communicator.

bool Initialized(void). Calls MPI_Init() and returns true if OOMPI_COMM_WORLD::Init() has already been called.

bool Is_null(void). Returns a boolean indicating whether the communicator is valid for use or not.

int Pack_size(OOMPI_Datatype type, int count). Calls MPI_Pack_size() with the datatype and count, and returns the calculated size.

int Pack_size(OOMPI_Message data). Calls MPI_Pack_size() with data and its internal count and returns the calculated size.
int Pack_size(OOMPI_Array message type, int count). Calls MPI_Pack_size() with type and the explicit count argument, and returns the calculated size.

virtual bool Test_inter(void) = 0. This is a pure virtual function which is defined in the classes which inherit from OOMPI_Comm. The derived classes call MPI_Comm_test_inter() and return true if the communicator is an inter-communicator, otherwise false.

Communicator Management

OOMPI_Compare Compare(OOMPI_Comm& a). Calls MPI_Comm_compare() and returns the result.

call bool operator==(OOMPI_Comm& a, OOMPI_Comm& b). Calls MPI_Comm_compare() and returns a logical true if the result is not OOMPI_UNEQUAL.

call bool operator!=(OOMPI_Comm& a, OOMPI_Comm& b). Calls MPI_Comm_compare() and returns a logical true if the result is OOMPI_UNEQUAL.

OOMPI_Group Group(void). Returns the OOMPI_Group of the communicator. This is analogous to the MPI_Group() function.

int Rank(void). Calls MPI_Comm_rank() and returns the rank of the calling process in the communicator.

int Size(void). Calls MPI_Comm_size() and returns the size of the communicator.

Error Handling

void Abort(int errorcode = 1). Calls MPI_Abort(). The actions of this function are MPI implementation dependent.

OOMPI_Error_action Get_error_action(void). Returns the default error action for the communicator.

void Set_error_action(OOMPI_Error_action action). Set the default error action associated with the communicator.

Port Access

OOMPI_Port operator[](int i). Returns the ith port in the communicator, where i is the rank of that port in the communicator.

Sendreceive

OOMPI_Status Sendrecv(SEND_BUF, int dest, int sendtag, RECV_BUF, int source, int recvtag). Call MPI_Sendrecv() with SEND_BUF and RECV_BUF.

OOMPI_Status Sendrecv_replace(BUFF, int dest, int sendtag, int source, int recvtag). Call MPI_Sendrecv_replace() with the BUFF argument.

See Also

OOMPI_Any_port, OOMPI_Cart_comm, OOMPI_Comm_world, OOMPI_Intercomm, OOMPI_Intra_comm, OOMPI_Graph_comm, OOMPI_Port

25
Name: OOMPI_Comm_world

Declaration:
```
#include "oompi.h"
class OOMPI_Comm_world
```

Description:
The OOMPI_Comm_world class has two primary functions which distinguish it from its base class (OOMPI_Intra_comm): MPI_Init() and MPI_Finalize(). There is only one instance of OOMPI_Comm_world, a global object named OOMPI_COMM_WORLD which has MPI_COMM_WORLD as its internal MPI communicator.

Inheritance:
This class is derived from OOMPI_Intra_comm.

Constructors/Destructors:

OOMPI_Comm_world(void). Default constructor. This constructor is invoked upon program startup and creates an empty communicator named OOMPI_COMM_WORLD. After invoking the Init() member function, OOMPI_COMM_WORLD will contain MPI_COMM_WORLD. There can only be one OOMPI_Comm_world instance; this constructor will fail if additional objects are instantiated.

OOMPI_Comm_world(const OOMPI_Comm_world& a). Copy constructor. This creates a communicator that contains MPI_COMM_NULL. OOMPI_COMM_WORLD is not allowed to be copied, but it can be duplicated into an OOMPI_Intra_comm with the Dup() member function.

OOMPI_Comm_world& operator=(const OOMPI_Comm_world& a). Assignment operator. Does nothing — OOMPI_COMM_WORLD is not allowed to be copied, but it can be duplicated into an OOMPI_Intra_comm with the Dup() member function.

~OOMPI_Comm_world(void). Destructor. Since the only valid instance of this class is global, the destructor is called when the program quits. However, it does not call MPI_Finalize() upon program completion if it was not explicitly invoked in the user program. It is the user’s responsibility to call the Finalize() function; if OOMPI_COMM_WORLD attempted to invoke MPI_Finalize() upon exiting the program, it is possible that the MPI implementation may have already shut down as part of the program exit procedures, and the invocation of MPI_Finalize() may cause an error.

Init / Finalize:

void Init(int& argc, char**& argv).
Calls MPI_Init(), and then invokes Init(void) (see below).

void Init(void) For thread safety, the body of this function is surrounded by a mutex so that only one thread may enter at a time. The first time this function is called, it adds MPI_COMM_WORLD to OOMPI_COMM_WORLD and creates all the ports. OOMPI datatypes and default tags may not have meaningful values until this function is invoked. Nothing will happen if Init() is called more than once. Additionally, since this function does not invoke MPI_Init(), it is safe to call this function to
initialize OOMPI even after MPI_Init() has been invoked from somewhere else in the user’s program. Invoking this function without first invoking MPI_Init() or Init(int& argc, char**& argv) is erroneous.

void Finalize(void). Calls MPI_Finalize(). No OOMPI function may be called after this function has been invoked.

Access and Information

bool Finalized(void). Returns a boolean indicating whether Finalize() has been invoked or not.

See Also

OOMPI_Comp, OOMPI_Intra_comm
**Name**  
OOMPI_Datatype

**Declaration**  
#include "oompi.h"  
class OOMPI_Datatype

**Description**  
An OOMPI_Datatype object is used to create user-defined datatypes. Access is provided to all the MPI-1 functions that are used to create and commit datatypes. The MPI_Datatype handle that is generated from the MPI_Type_*() calls is contained within the OOMPI_Datatype object.

The OOMPI_Datatype object is meant to be a static attribute of all user objects that will be passed as messages. Therefore, the datatype only needs to be created once for the entire class (see the Built() member function below). The constructors for the user object should check the Built() function to see if the datatype has been built yet. If not, they should call the proper member functions to build the datatype. See the example in Section 3.4. **Warning:** Note that if a datatype has been created in an OOMPI_Datatype object, another cannot be created in the same object. Undefined results will occur if this rule is violated.

**Inheritance**  
This class is derived from OOMPI_Tag.

**Constructors/Destructors**  

OOMPI_Datatype(MPI_Datatype type = MPI_DATATYPE_NULL, int tag = OOMPI_MPI_DATATYPE_TAG). MPI constructor. Will create a reference to MPI_-DATATYPE_NULL if not invoked with a valid MPI_Datatype. The typical usage is to instantiate an OOMPI_Datatype with no arguments and then proceed to create the datatype with the functions listed below. This constructor also serves as the default constructor.


OOMPI_Datatype &operator=(const OOMPI_Datatype &a). Assignment operator. Perform a shallow (reference counted) assignment. The current reference to the internal MPI_Datatype is deleted (which may trigger a call to MPI_Type_free()).

virtual ~OOMPI_Datatype(void). Destructor. Delete the current reference to the internal MPI_Datatype (which may trigger a call to MPI_Type_free()).

**Access and Information**  

bool Built(void). Returns true if the internal datatype has been built. This function should be checked in the constructors of user objects that will have corresponding MPI datatypes built.
**Discussion:** Note that this function performs a down operation on a mutex to ensure that the datatype-building process is thread safe. The corresponding up operation is immediate if the datatype has been built. Otherwise, the up is performed when the datatype is built and committed.

`MPI_Datatype& Get_mpi(void).` Returns a reference to the internal `MPI_Datatype`.

`OOMPI_Aint Extent(void).` Calls `MPI_Type_extent` with the datatype of the instance of this object and returns the extent.

`bool Is_null(void).` Returns a boolean indicating whether the `OOMPI_Datatype` is valid to use or not.

`MPI_Aint Lb(void).` Calls `MPI_Type_lb` with the datatype of the instance of this object and returns the lower bound.

`int Size(void).` Calls `MPI_Type_size` with the datatype of the instance of this object and returns the size.

`MPI_Aint Ub(void).` Calls `MPI_Type_ub` with the datatype of the instance of this object and returns the upper bound.

**“Simple” Datatype Building**

`void Contiguous_type(OOMPI_Datatype type, int count).` \(^4\)

`void Contiguous(OOMPI_Message type, int count).`\(^4\)

`void Contiguous(OOMPI_Array_message type, int count).`\(^4\)

Calls `MPI_Type_contiguous()`. The argument type is used only to extract its type, not its count or location. After the `MPI_Type_contiguous()` function is called, `MPI_Type_commit()` is called so that this instance of `OOMPI_Datatype` is ready for use.

`void Hindexed_type(int blocklengths[], OOMPI_Aint disps[], OOMPI_Datatype type, int count).`\(^4\)

`void Hindexed(int blocklengths[], OOMPI_Aint disps[], OOMPI_Message type, int count).`\(^4\)

`void Hindexed(int blocklengths[], OOMPI_Aint disps[], OOMPI_Array_message type, int count).`\(^4\)

Calls `MPI_Type_hindexed()`. The argument type is used only to extract its type,

\(^4\)Notice that the first function of this series has an additional `_type` suffix added to its name. The reason for this is to prevent a potential ambiguity in some MPI implementations. If the underlying MPI declares `MPI_DATATYPE` to be an int (i.e. an integer-based handle), passing an integer as the data argument to these functions could resolve to both the MPI constructor of `OOMPI_Datatype` and the integer promotions for `OOMPI_Message`. Therefore, it was decided to change the signature of one of the functions by adding `_type` to the function name. This has been done to all the functions in this section that take an `OOMPI_Datatype` instance as the data argument.
not its count or location. After the `MPI_Type_hindexed()` function is called, `MPI_Type_commit()` is called so that this instance of `OOMPI_Datatype` is ready for use.

```c
void Hvector_type(int blocklength, int stride, OOMPI_Datatype type, int count).
void Hvector(int blocklength, int stride, OOMPI_Message type, int count).
void Hvector(int blocklength, int stride, OOMPI_Array message type, int count).
```
Calls `MPI_Type_hvector()`. The argument `type` is used only to extract its type, not its count or location. After the `MPI_Type_hvector()` function is called, `MPI_Type_commit()` is called so that this instance of `OOMPI_Datatype` is ready for use.

```c
void Indexed_type(int blocklengths[], int disps[], OOMPI_Datatype type, int count).
void Indexed(int blocklengths[], int disps[], OOMPI_Message type, int count).
void Indexed(int blocklengths[], int disps[], OOMPI_Array message type, int count).
```
Calls `MPI_Type_indexed()`. The argument `type` is used only to extract its type, not its count or location. After the `MPI_Type_indexed()` function is called, `MPI_Type_commit()` is called so that this instance of `OOMPI_Datatype` is ready for use.

```c
void Vector_type(int blocklength, int stride, OOMPI_Datatype type, int count).
void Vector(int blocklength, int stride, OOMPI_Message type, int count).
void Vector(int blocklength, int stride, OOMPI_Array message type, int count).
```
Calls `MPI_Type_vector()`. The argument `type` is used only to extract its type, not its count or location. After the `MPI_Type_vector()` function is called, `MPI_Type_commit()` is called so that this instance of `OOMPI_Datatype` is ready for use.

### "Complex" Datatype Building

```c
void Struct_start(void *t, void *lb = 0). Start defining a complex data structure. t must be the this pointer from the source object so that displacements can be calculated. If the lb parameter is specified, it is set as the MPI LB of the structure. After this function, multiple calls to operator<<( ) and/or Entry() are made to specify the type, count, and offset of the individual members in the object.

OOMPI_Datatype& operator<<(OOMPI_Message data). Promote a variable to obtain and store its type, count, and offset (from a call to MPI_Address( )).```
void Entry(DATA). Promote a variable to obtain and store its type, count, and offset (from a call to MPI_Address()).

void Struct_end(void *ub = 0). Assemble all the information from the previous calls to operator<<() and Entry(). If the ub parameter is specified, it is set as the MPI_UB of the structure. Then the functions MPI_Type_struct() and MPI_Type_commit() are invoked.

See Also OOMPI_Array_message, OOMPI_Message, OOMPI_User_type
The OOMPI_Environment class is used to group all environment-related functions into one class. There is only one instance of the OOMPI_Environment class, OOMPI_ENV. It is globally scoped. Attempts to copy or instantiate other OOMPI_Environment instances will fail.

Inheritance
None.

Buffer
void Buffer_attach(int size). Allocates a buffer of size size and calls MPI_Buffer_attach().

void Buffer_attach(void *buffer, int size). Calls MPI_Buffer_attach() with the arguments buffer and size.

int Buffer_detach(void). Calls MPI_Buffer_detach() and returns the size of the buffer that was detached.

Processor Dependant
char* Get_processor_name(void). Returns a newly allocated string filled with processor's name as returned by MPI_Get_processor_name().

char* Get_processor_name(int& len). Similar to the previous function, except that it additionally returns the length of the string in len.

void Get_processor_name(char name[], int& len). Call MPI_Get_processor_name() and fill name and len with the result. name must be at least OOMPI_MAX_PROCESSOR_NAME characters long.

double Wtick(void). Returns the result of MPI_Wtick().

double Wtime(void). Returns the result of MPI_Wtime().

Profiling
int Pcontrol(int level, ...). Calls MPI_Pcontrol() and returns the result.

See Also
Name: OOMPI_Error

Declaration:

```c
#include "oompi.h"
class OOMPI_Error
```

Description:
The OOMPI_Error class is used for throwing and catching exceptions. When an MPI error is detected, if the OOMPI_ERROR_ACTION on the communicator on which the error occurred is set to OOMPI_ERRORS_EXCEPTION, an OOMPI_Error instance is constructed and thrown. Errors that occur that are not associated with any communicator are thrown on OOMPI_COMM_WORLD.

Inheritance:
None.

Access and Information:

- `OOMPI_Comm& Get_comm(void)`. Returns the communicator associated with the error.
- `int Get_code(void)`. Returns the error code associated with the error.
- `int Get_class(void)`. Returns the error class associated with the error.
- `char *Get_string(void)`. Returns a pointer to a newly allocated string containing the MPI implementation-defined error string.
- `void Get_string(char msg[], int &len)`. Fill `msg` and `len` with the MPI implementation-defined error string and its length, respectively. `msg` must be at least OOMPI_MAX_ERROR_STRING characters long.

See Also:
OOMPI_Comm, OOMPI_Comm_world
Name: OOMPI_Graph_comm

Declaration:
```
#include "oompi.h"
class OOMPI_Graph_comm
```

Description:
A class for MPI communicators with graph topology.

Inheritance:
This class is derived from OOMPI_Intra_comm.

Constructors/Destructors:
- `OOMPI_Graph_comm(MPI_Comm comm = MPI_COMM_NULL)` - MPI constructor. If `comm` is a valid communicator with an associated graph topology, the constructor for the OOMPI_Comm base class is invoked with `comm`. Otherwise, it is invoked with `MPI_COMM_NULL`.
- `OOMPI_Graph_comm(const OOMPI_Graph_comm& a)` - Copy constructor. The copy constructor for the OOMPI_Comm base class is invoked.
- `OOMPI_Graph_comm& operator=(const OOMPI_Graph_comm& a)` - Assignment operator. The assignment operator for the OOMPI_Comm base class is invoked.
- `OOMPI_Graph_comm(const OOMPI_Intra_comm& intra_comm, int nnodes, int index[], int edges[], bool reorder = false)` - Constructor. Calls `MPI_Graph_create()` to create a new communicator with a graph topology.
- `~OOMPI_Graph_comm()` - Destructor. Destruction is handled by the OOMPI_Comm base class.

Communicator Management:
- `OOMPI_Graph_comm Dup(void)` - Returns a new OOMPI_Graph_comm that was created with the result of `MPI_Comm_dup()`.

Graph Topology Functions:
- `void Dims_get(int& nnodes, int& nedges)` - Calls `MPI_Graphdims_get()`. The number of nodes and edges are returned in `nnodes` and `nedges`, respectively.
- `int Num_nodes(void)` - Shortcut function that invokes `MPI_Graphdims_get()` and returns the number of nodes.
- `int Num_edges(void)` - Shortcut function that invokes `MPI_Graphdims_get()` and returns the number of edges.
- `void Get(int maxindex, int maxedges, int index[], int edges[])` - Calls `MPI_Graph_get()` to obtain information about the graph. If `maxindex/maxedges` is 0, `Num_nodes()`/`Num_edges` is used.
void Get(int index[], int edges[]). Calls MPI_Graph_get() to obtain information about the graph. Num_nodes() and Num_edges() are used for maxindex and maxedges, respectively.

void Get_edges(int edges[]). Shortcut for the previous function, except it only fills the edges array.

int *Get_edges(void). Shortcut for the previous function, except that it allocates, fills, and returns the edges array.

void Get_index(int index[]). Shortcut for the function listed above, except that it only fills the index array.

int *Get_index(void). Shortcut for the previous function, except it allocates, fills, and returns the index array.

int *Neighbors(int maxneighbors, int rank, int neighbors[]). Calls MPI_Graph_neighbors(). The value of neighbors is returned for the given rank.

int *Neighbors(int rank, int neighbors[]). Shortcut for the previous function, except Neighbors_count() is used for maxneighbors.

int *Neighbors(int neighbors[]). Shortcut for the previous function, except the current rank number is used for rank.

int *Neighbors(int maxneighbors, int rank). Shortcut for the function listed above, except the neighbors array is allocated, filled, and returned.

int *Neighbors(int rank). Shortcut for the previous function, except Neighbors_count() is used for maxneighbors.

int *Neighbors(void). Shortcut for the previous function, except Neighbors_count() is used for maxneighbors, and the current rank is used for rank.

int Neighbors_count(int rank). Calls MPI_Graph_neighbors_count() and returns the number of neighbors for the specified rank.

int Neighbors_count(void). Calls MPI_Graph_neighbors_count() and returns the number of neighbors for the current rank.

See Also OOMPI_Cart_comm, OOMPI_Comm, OOMPI_Intra_comm
Name                OOMPI_Group

Declaration          #include "oompi.h"
                    class OOMPI_Group

Description          A class for MPI group management.

Inheritance          None.

Constructors/Destructors

OOMPI_Group(MPI_Group group = MPI_GROUP_NULL). MPI constructor. If group is a valid group, a new container is created for group and a reference to it is made. This constructor also acts as the default constructor.


~OOMPI_Group(). Destructor. The destructor deletes its reference, which may result in a call to MPI_Group_free().

Access and Information

MPI_Group& Get_mpi(). Returns a reference to the internal MPI_Group.

bool Is_empty(void). Returns a boolean indicating whether the underlying group is MPI_GROUP_EMPTY or not.

bool Is_null(void). Returns a boolean indicating whether the instance is valid to use or not.

int Rank(void). Invokes MPI_Group_rank() to return the rank of the calling process in the group.

int Size(void). Invokes MPI_Group_size() to return the number of processes in the group.

void Translate_ranks(int n, int ranks1[], OOMPI_Group g2, int ranks2[]). This function is used to determine the relative numbering of the same processes in two different groups by calling MPI_Group_translate_ranks(). ranks1 and ranks2 are filled.

int *Translate_ranks(int n, int ranks1[], OOMPI_Group& g2). Shortcut for the previous function, except that ranks2 is allocated, filled, and returned.
Operators

OOMPI_Compare Compare(OOMPI_Group group). Performs a comparison with group by an invocation to MPI_Group_compare() and returns the result.

friend bool operator==(OOMPI_Group g1, OOMPI_Group g2). This operator invokes MPI_Group_compare() returns a true if g1 and g2 are not MPI_UNEQUAL.

friend bool operator!=(OOMPI_Group g1, OOMPI_Group g2). Invokes MPI_Group_Compare() and returns true if g1 and g2 are MPI_UNEQUAL.

friend OOMPI_Group operator|(OOMPI_Group g1, OOMPI_Group g2). Invokes MPI_Group_union() and returns a new OOMPI_Group that is the union of the groups g1 and g2.

friend OOMPI_Group operator&(OOMPI_Group g1, OOMPI_Group g2). Invokes MPI_Group_intersection() and returns a new OOMPI_Group that is the intersection of the groups g1 and g2.

friend OOMPI_Group operator-(OOMPI_Group g1, OOMPI_Group g2). Invokes MPI_Group_difference() and returns a new OOMPI_Group that is the difference of the groups g1 and g2.
Inclusion/Exclusion

OOMPI_Group Excl(int n, int ranks[]). This function calls MPI_Group_excl() and returns a new OOMPI_Group.

OOMPI_Group Incl(int n, int ranks[]). This function calls MPI_Group_incl() and returns a new OOMPI_Group.

OOMPI_Group Range_excl(int n, int ranges[][3]). This function calls MPI_Group_range_excl() and returns a new OOMPI_Group.

OOMPI_Group Range_incl(int n, int ranges[][3]). This function calls MPI_Group_incl() and returns a new OOMPI_Group.

See Also OOMPI_Comm
Name        OOMPI_Inter_comm

Declaration  
#include "oompi.h"
#include "oompi.h"

class OOMPI_Inter_comm

description  A class for MPI intercommunicators.

Inheritance  This class is derived from OOMPI_Comm.

Constructors/Destructors

OOMPI_Inter_comm(MPI_Comm comm = MPI_COMM_NULL). MPI constructor. If comm is a valid intercommunicator, the constructor for the OOMPI_Comm base class is invoked with comm. Otherwise, it is invoked with MPI_COMM_NULL.

OOMPI_Inter_comm(const OOMPI_Inter_comm& a). Copy constructor. The copy constructor for the OOMPI_Comm base class is invoked.

OOMPI_Inter_comm& operator=(const OOMPI_Inter_comm& a). Assignment operator. The assignment operator for the OOMPI_Comm base class is invoked.

OOMPI_Inter_comm(OOMPI_Intra_comm& local_comm, int local_leader, OOMPI_Intra_comm& peer_comm, int remote_leader, int tag = OOMPI_INTERCOMM_CREATE_TAG). Constructor. Uses MPI_Intercomm_create() to construct an MPI intercommunicator between local_comm and remote_comm.

virtual ~OOMPI_Inter_comm(). Destructor. Destruction is handled by the OOMPI_Comm base class.

Communicator Management

OOMPI_Inter_comm Dup(void). Returns a new OOMPI_Inter_comm that was created with the result of MPI_Comm_dup().

Intercommunicator Functions

OOMPI_Intra_comm Merge(bool high = true). Calls MPI_Intercomm_merge() to create a new MPI communicator and returns the newly created OOMPI_Intra_comm. high indicates relative ordering of the two intra-communicators.

OOMPI_Group Remote_group(void) Returns an OOMPI_Group that holds the MPI_Group obtained by calling MPI_Comm_remote_group().

int Remote_size(void). Returns the result of MPI_Comm_remote_size() of the remote communicator.

See Also  OOMPI_Comm OOMPI_Intra_comm,
Name: OOMPI_Intra_comm

Declaration: 
#include "oompi.h"
class OOMPI_Intra_comm

Description: A class for MPI intracommunicators.

Inheritance: This class is derived from OOMPI_Comm.

Constructors/Destructors

OOMPI_Intra_comm(MPI_Comm comm = MPI_COMM_NULL). MPI constructor. If comm is a valid intracommunicator, the constructor for the OOMPI_Comm base class is invoked with comm. Otherwise, it is invoked with MPI_COMM_NULL and the created OOMPI_Cart_comm is equivalent to OOMPI_COMM_NULL.

OOMPI_Intra_comm(const OOMPI_Intra_comm &a). Copy constructor. The copy constructor for the OOMPI_Comm base class is invoked.

OOMPI_Intra_comm &operator=(const OOMPI_Intra_comm &a). Assignment operator. The assignment operator for the OOMPI_Comm class is invoked.

virtual ~OOMPI_Intra_comm(). Destructor. Destruction is handled by the OOMPI_Comm base class.
Communicator Management

```c
OMPI_Intra_comm Dup(void). Returns a new OOMPI_Intra_comm that was
created with the result of MPI_Comm_dup().
```

Groups, Contexts, and Communicators

```c
OMPI_Intra_comm Create(OMPI_Group& group). Calls MPI_Comm_create() to obtain a new MPI communicator based on the internal MPI group of
group. A new OOMPI_Intra_comm constructed from this MPI communicator is
returned.
```

```c
OMPI_Intra_comm Split(int color, int key = 0). Calls MPI_Comm_split() to obtain a new MPI communicator based on the internal MPI group of
group. A new OOMPI_Intra_comm constructed from this MPI communicator is
returned.
```

Collective Communication

```c
void Allgather(SENDBUF, RECVBUF). Calls MPI_Allgather().
```

```c
void Allgatherv(SENDBUF, OOMPI_Array_message recvbuf, int recvcounts[],
int displs[]). Calls MPI_Allgatherv(). Note that the recvbuf argument
must be an OOMPI_array_message, because an array of receive counts is neces-
sary, not a single receive count.
```

```c
void Allreduce(SENDBUF, OOMPI_Message recvbuf, const OOMPI_Op& op). Calls MPI_Allreduce(), using the internal MPI_Op of op. Note that
only two variations of this function are given, because there is only one count argument for both choice arguments; it does not make sense to mix and match (which count argument would be used?).
```

```c
void Alltoall(SENDBUF, RECVBUF). Calls MPI_Alltoall().
```

```c
void Alltoallv(OMPI_Array_message sendbuf, int sendcounts[],
int sdispls[], OOMPI_Array_message recvbuf, int recvcounts[],
int rdispls[]). Calls MPI_Alltoallv().
```

```c
void Barrier(void). Calls MPI_Barrier().
```

```c
void Reduce_scatter(SENDBUF, OOMPI_Array_message recvbuf, int
recvcounts[], const OOMPI_Op& op). Calls MPI_Reduce_scatter().
```

```c
void Scan(Sendbuf, OOMPI_Array_message recvbuf, const OOMPI_Op& op). Calls MPI_Scan().
```
MPI_ANY_SOURCE Functions

OOMPI_Intra_comm& operator>>(OOMPI_Message data). Invokes an MPI_Recv() with MPI_ANY_SOURCE as the source argument. The result is put into data. The default tag for OOMPI_Message is used.

OOMPI_StatusRecv(RECVBUF, int tag = OOMPI_NO_TAG). Invokes an MPI_Recv() with MPI_ANY_SOURCE as the source argument. The result is put into data. If tag is not specified, the default tag for data is used.

OOMPI_Request Irecv(RECVBUF, int tag = OOMPI_NO_TAG). Invokes an MPI_Irecv() with MPI_ANY_SOURCE as the source argument. The result is put into data. If tag is not specified, the default tag for data is used.

OOMPI_Request Recv_init(RECVBUF, int tag = OOMPI_NO_TAG). Invokes an MPI_Recv_init() with MPI_ANY_SOURCE as the source argument. The result is put into data. If tag is not specified, the default tag for data is used.

OOMPI_Status Probe(int tag). Invokes MPI_Probe() with MPI_ANY_SOURCE as the source argument.

OOMPI_Status Iprobe(int tag, bool& flag). Invokes MPI_Iprobe() with MPI_ANY_SOURCE as the source argument.

bool Iprobe(int tag). Shortcut for the previous function, except it returns the value of flag.

General Topology Functions

int Cart_map(int ndims, int dims[], bool periods[]). Invokes MPI_Cart_map() and returns the resulting newrank.

int Cart_map(int ndims, int dims[], bool periods). Shortcut function similar to the above function, except periods is a single bool indicating the periodicity of all dimensions.

int *Dims_create(int ndims, int dims[], int nnodes = 0). Calls MPI_Dims_create(). Returns the dimensions array. If nnodes is not specified, Size() is used.

int Graph_map(int ndims, int dims[], int edges[]). Invokes MPI_Graph_map() and returns the resulting newrank.

See Also

OOMPI_Cart_comm, OOMPI_Comm, OOMPI_Inter_comm, OOMPI_Graph_comm
Name: OOMPI_Message

Declaration:
```
#include "oompi.h"
class OOMPI_Message
```

Inheritance:
This class inherits from OOMPI_Tag.

Description:
A class for managing the data associated with a message; its type, count, tag, and data. This class also promotes the base types into OOMPI_Message objects so that they can be easily accessed in other OOMPI functions.

OOMPI_Message objects are usually created via promotion (and immediately destroyed) to keep the calling semantics of OOMPI simple. However, sometimes it is desirable to explicitly create an OOMPI_Message. Explicit creation of OOMPI_Message objects allows the default type tag to be overridden. OOMPI_Message objects can be explicitly created for scalar variables and contiguous arrays that require a count argument.

If an OOMPI_Message is explicitly created, it can be re-used even if the value of the variable (or values in an array) change. The OOMPI_Message object keeps a pointer to the data, not a copy of the data. **NOTE:** If the data pointed to by an OOMPI_Message object is deleted, the OOMPI_Message will still reference the deleted memory. It is considered erroneous to attempt to use the OOMPI_Message after the data memory has been deleted.

Constructors/Destructors


OOMPI_Message(<TYPE> data).
OOMPI_Message(<TYPE>& data, int tag) Constructors. These constructors are used to promote a base type into an OOMPI_Message type. These functions are *not* templated; the <TYPE> notation is used for brevity. In the above function prototypes <TYPE> can take on any of the following base types:

<table>
<thead>
<tr>
<th>char</th>
<th>unsigned char</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>unsigned short</td>
<td>double</td>
</tr>
<tr>
<td>int</td>
<td>unsigned</td>
<td>OOMPI_Packed</td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The *tag* argument sets the default tag for that message. If the *tag* is not specified, the default is used (depending on the type).
OOMPI_Message(<TYPE> *data, int count)
OOMPI_Message(<TYPE> *data, int count, int tag). Constructors. Constructors for explicitly creating an array message (not to be confused with an OOMPI_Array_message). These functions are used when it is desirable to either override the default tag for a base type, or create an envelope for an array that can be used (for example) in the streams send/receive interface. <TYPE> supports the same types as listed above.

OOMPI_Message(const OOMPI_Datatype& type, void *top, int count = 1). Constructor. Used for explicitly creating OOMPI_Message objects based upon OOMPI_Datatypes. Uses a default tag based upon type.

OOMPI_Message(const OOMPI_Datatype& type, void *top, int count, int tag). Constructor. Used for explicitly creating OOMPI_Message objects based upon OOMPI_Datatypes.

Access Functions

void *Get_top(void). This access function returns the address of the top of the object that is being encapsulated in the OOMPI_Message. Since the OOMPI_Message retains a pointer to the data, it will always reflect the current value of the data (even if it changes after the OOMPI_Message was created). This is useful for creating “persistent” OOMPI_Messages that can be re-used in successive OOMPI calls.

MPI_Datatype Get_type(void). This access function returns the MPI_Datatype associated with the message. This function is only meant to be used by OOMPI; it is not considered to be part of the user interface.

int Get_count(void). Returns the count of the current object (will always be 1 for data that was promoted).

See Also OOMPI Constants, OOMPI_Datatype, OOMPI_Intra_comm, OOMPI_Packed, OOMPI_Port
Name: OOMPI_Op

Declaration:
```
#include "oompi.h"
class OOMPI_Op
```

Description: An OOMPI_Op object has four main functions: construction, copying, use as an argument for reduction operations, and destruction.

Inheritance: None.

Constructors/Destructors:

- `OOMPI_Op(MPI_Op op = MPI_Op_NULL)`. MPI constructor. A new container is created for `op` and a reference to it is made. This constructor also acts as the default constructor.


- `OOMPI_Op(MPI_User_function *function, bool commutative = true)`. Constructor. Calls `MPI_Op_create()` to create an `MPI_Op` handle. A new container is created for the resulting `MPI_Op` handle and a reference to it is made.

Discussion: It would seem more consistent to have an `OOMPI_User_function` user's callback method, prototyped as:
```
typedef void OOMPI_User_function&(void *invec, void *inoutvec, int &len, OOMPI_Datatype &datatype);
```

The `OOMPI` version of the user's callback method would have the same semantics as the corresponding `MPI` callback function, except that it would provide an `OOMPI_Datatype` rather than an `MPI_Datatype`. Additionally, the callback method can be a member function of a user object than can have local data associated with it.

Such a callback scheme would necessitate a level of indirection, where `OOMPI` registers an intermediate `MPI` callback function that can provide the translation from the `MPI_Datatype` to the corresponding `OOMPI_Datatype`, and then invoke the user callback method. However, it seems that the only reasonable way to do this would be to wrap the calls to `MPI_Reduce()` and `MPI_Reduce_scatter()` in functions that cache attributes (or other global data) containing the relevant user method pointer and `OOMPI_Datatype`. This may not provide good performance.

For this release of `OOMPI`, it was decided that `MPI` user-defined operators are low-level func-
tions, and therefore must use the corresponding MPI_Datatype. **NOTE:** It is erroneous to generate an OOMPI_Datatype from the MPI_Datatype that is passed to the user's callback function. This is erroneous for the same reasons that it is erroneous to create a new OOMPI object with the result of a Get_mpi() function (see the Compatibility paragraph in Section 3.12).

\~OOMPI_Op(). Destructor. The destructor deletes the reference to the MPI_Op handle (which may trigger a call to MPI_Op_free())

**Access and Information**

bool Is_null(void). Returns a boolean indicating whether the instance is valid for use in reduction operations or not.


**See Also**

OOMPI_Comm, OOMPI_Intra_Comm
Name | OOMPI_Packed  
---|---
Declaration | 
#include "oompi.h"
class OOMPI_Packed  
Description | The OOMPI_Packed object is used to provide a streams-based interface to the MPI-1 packing and unpacking functions. Note that access to MPI_Pack_size() is not provided through this object. Instead, it is provided through the OOMPI_Comm object. 

**Discussion:** Since MPI_Pack_size() is used to determine how large a buffer is necessary to create in order to pack or unpack an message, it is pointless to create an OOMPI_Packed object with a buffer only to determine what the “real” size of the buffer should be. Therefore, MPI_Pack_size() is encapsulated in all communicators so that the proper buffer size can be determined before an OOMPI_Packed object is created.  

Inheritance | This class inherits functions from OOMPI_Tag.  
Constructors/Destructors | 
OOMPI_Packed(int size, OOMPI_Comm &c, int tag = OOMPI_PACKED_TAG). Constructor. A buffer of length size is allocated for packing and unpacking. The tag will be used as the default tag in the streams-based interface for sending and receiving this object. 

OOMPI_Packed(void *ptr, int size, OOMPI_Comm &c, int tag = OOMPI_PACKED_TAG). Constructor. A buffer of length size is provided by the caller. The tag will be used as the default tag in the streams-based interface for sending and receiving this object. 

OOMPI_Packed(const OOMPI_Packed &a). Copy constructor. The copy constructor performs a deep copy of the object (the entire buffer is copied).  

OOMPI_Packed &operator=(const OOMPI_Packed &a). The assignment operator performs a deep copy of the a object (the entire buffer is copied). The destination buffer is only deleted if the OOMPI_Packed object initially created it; if the user specified the buffer in the OOMPI_Packed constructor, it is not deleted before copy takes place (but OOMPI will allocate a new buffer for the destination of the copy). 

~OOMPI_Packed(). Destructor. The destructor deletes the buffer only if the OOMPI_Packed object created the buffer; if the user specified the buffer in the OOMPI_Packed constructor, it is not deleted.  

Access and Information | 
int Get_position(void). Returns the current position of the pack/unpack.
int Get_size(void). Returns the size of the buffer available for packing and unpacking.

void Reset(void). Resets the state of the object back to the beginning of the buffer.

int Set_position(int size). Sets the current position of the pack/unpack. Returns the actual position that is set.

int Set_size(int size). Allows the user to expand or shrink the buffer used for packing and unpacking. Returns the size that the buffer is actually set to.

Packing and Unpacking

void Start(int position = 0). Resets the state of the object and prepares for repeated calls to operator<<() (to pack into the buffer) or operator>>() (to unpack from the buffer). Optionally specify a specific location to start in the buffer.

OOMPI_Packed& operator<<(OOMPI_Message data). Pack the specified object into the buffer using MPI_Pack(). Attempting to pack beyond the end of the buffer is undefined; it is the user’s responsibility to ensure that this does not happen.

void Pack(BUFFER). Pack the specified object into the buffer using MPI_Pack(). Attempting to pack beyond the end of the buffer is undefined; it is the user’s responsibility to ensure that this does not happen.

OOMPI_Packed& operator>>(OOMPI_Message data). Unpack the specified object from the buffer using MPI_Unpack(). Attempting to unpack beyond the end of the buffer is undefined; it is the user’s responsibility to ensure that this does not happen.

void Unpack(BUFFER). Unpack the specified object into the buffer using MPI_−Unpack(). Attempting to unpack beyond the end of the buffer is undefined; it is the user’s responsibility to ensure that this does not happen.

See Also OOMPI_Comm, OOMPI_Intra_comm, OOMPI_Port, OOMPI_Tag
Name  OOMPI_Port

Declaration  #include "oompi.h"
class OOMPI_Port

Description  A class for managing point to point and rooted collective operations.

Inheritance  None.

Constructors/Destructors

OOMPI_Port(void). Default constructor. Sets the internal communicator to MPI_COMM_NULL and the internal rank to OOMPI_PROC_NULL.

OOMPI_Port(MPI_Comm c, int my_rank). MPI constructor. Create an OOMPI_Port in the corresponding MPI_Comm with the specified rank my_rank.


~OOMPI_Port(). Destructor. Delete the reference to the internal MPI_Comm. This may trigger a call to MPI_Comm_free() if the original OOMPI_Comm has been deleted.

Access and Information

int Rank(void). Calls MPI_Rank() to return the rank of the port in its communicator.

Intercommunicator Management

OOMPI_Inter_comm Intercomm_create(OOMPI_Intra_Comm& peer_comm, int remote_leader, int tag = OOMPI_INTERCOMM_CREATE_TAG). Calls MPI_Intercomm_create() to create a new intercommunicator. The local leader is implicitly specified by the invoking OOMPI_Port instance. The remote leader is specified as an (peer_comm, remote_leader) pair.

OOMPI_Inter_comm Intercomm_create(OOMPI_Port& peer_port, int tag = OOMPI_INTERCOMM_CREATE_TAG). Calls MPI_Intercomm_create() to create a new intercommunicator. The local leader is implicitly specified by the OOMPI_Port that the function is invoked on. The remote leader is specified with peer_port.
Rooted Collective Operations

```cpp
// Rooted Collective Operations

void Bcast(BUF). Calls MPI_Bcast() with the choice argument BUF.

void Gather(SENDBUF, RECVBUF). Calls MPI_Gather() with the choice arguments SENDBUF and RECVBUF.

void Gather(SENDBUF). Shortcut for the previous function; non-root processes may call this function, and therefore not have to specify the RECVBUF.

void Gatherv(SENDBUF, OOMPI_Array_message recvbuf, int recvcounts[], int displs[]). Calls MPI_Gatherv() with SENDBUF and recvbuf.

void Gatherv(SENDBUF). Shortcut for the previous function; non-root processes may call this function, and therefore not have to specify the RECVBUF.

void Reduce(SENDBUF, RECVBUF, const OOMPI_Op& op). Calls MPI_Reduce(), using the internal MPI_Op of op.

void Reduce(SENDBUF, const OOMPI_Op& op). Shortcut for the previous function; non-root processes may call this function, and therefore not have to specify the RECVBUF.

void Scatter(SENDBUF, RECVBUF). Calls MPI_Scatter() with the choice arguments SENDBUF and RECVBUF.

void Scatter(RECVBUF). Shortcut for the previous function; non-root processes may call this function, and therefore not have to specify the SENDBUF.

void Scatterv(OOMPI_Array_message sendbuf, int recvcounts[], int displs[], RECVBUF). Calls MPI_Scatterv() with sendbuf and RECVBUF.

void Scatterv(RECVBUF). Shortcut for the previous function; non-root processes may call this function, and therefore not have to specify the SENDBUF.
```

Streams Interface

```cpp
// Streams Interface

OMPI_Port& operator<<(OMPI_Message buf). Streams interface to MPI_Send().

OMPI_Port& operator>>(OMPI_Message buf). Streams interface to MPI_Recv().
```

Sends

```cpp
// Sends

void Bsend(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Bsend() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Bsend_init(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Bsend_init() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.
```
void Ibsend(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Ibsend() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Irsend(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Irsend() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Isend(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Isend() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Issend(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Issend() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Rsend(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Rsend() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Rsend_init(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Rsend_init() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Send(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Send() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Send_init(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Send_init() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Ssend(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Ssend() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

void Ssend_init(SENDBUF, int tag = OOMPI_NO_TAG). Calls MPI_Ssend_init() with the choice argument SENDBUF. If no tag is specified, the default tag for SENDBUF is used.

Receives

OOMPI_Request Irecv(RECVBUF, int tag = OOMPI_NO_TAG). Calls MPI_Irecv() with the choice argument RECVBUF. If no tag is specified, the default tag for RECVBUF is used.

OOMPI_StatusRecv(RECVBUF, int tag = OOMPI_NO_TAG). Calls MPI_Rrecv() with the choice argument RECVBUF. If no tag is specified, the default tag for RECVBUF is used.
OOMPI_Request Recv_init(RECVBUF, int tag = OOMPI_NO_TAG). Calls MPI_Recv_init() with the choice argument RECVBUF. If no tag is specified, the default tag for RECVBUF is used.

See Also
OOMPI_Array_message, OOMPI_Comm, OOMPI_Intra_comm, OOMPI_Inter-comm, OOMPI_Message, OOMPI_Request, OOMPI_Status, OOMPI_User_type
**Name**
OOMPI_Request

**Declaration**
#include "oompi.h"
class OOMPI_Request

**Inheritance**
None.

**Description**
A class for encapsulating a single MPI_Request handle and its associated functionality.

**Constructors/Destructors**

OOMPI_Request(MPI_Request request = MPI_REQUEST_NULL). MPI constructor. A new container is created for request and a reference to it is made. This constructor also acts as the default constructor.

OOMPI_Request(const OOMPI_Request& a). Copy constructor. Perform a deep copy; OOMPI_Request objects are not reference counted.

OOMPI_Request& operator=(const OOMPI_Request& a). Assignment operator. Perform a deep copy; OOMPI_Request objects are not reference counted.

OOMPI_Request& operator=(const MPI_Request& a). Assignment operator. Perform a deep copy; OOMPI_Request objects are not reference counted.

~OOMPI_Request(void). Free the memory associated with the current MPI_Request. Does not trigger a call to MPI_Request_free(). This must be done explicitly by the user using the Free() method if the request is persistent.

**Access and Information**

bool Is_null(void). Return a boolean indicating whether the underlying MPI_Request is MPI_REQUEST_NULL or not.

MPI_Request& Get_mpi(void). Returns the internal MPI_Request.

bool operator==(const OOMPI_Request& a) Returns a boolean indicating whether the current request refers to the same MPI_Request as a.

bool operator!=(const OOMPI_Request& a) Returns a boolean indicating whether the current request does not refer to the same MPI_Request as a.

**Test / Wait**

OOMPI_Status Test(bool& flag). Calls MPI_Test(). If the underlying MPI request is valid (i.e., it is not MPI_REQUEST_NULL) and the operation on the current MPI request has completed, flag is set to true. An OOMPI_Status object is returned.

bool Test(oompi::Status& status). Shortcut for the previous function; returns the boolean flag.

53
OOMPI_Status Wait(void). Calls MPI_Wait() and waits until the communication associated with the internal MPI request has completed. An OOMPI_MPI_Status object is returned.

Start / Cancel

void Cancel(). Calls MPI_Cancel() to cancel the current communication on the MPI_Request handle.

void Start(). Calls MPI_Start() to initiate non-blocking communication on the MPI_Request handle.

Free

void Free(). Calls MPI_Request_Free() to free this request if it is non-null. This request is a no-op on POE 2.1.0.14 because of a bug in this implementation’s version of MPI_Request_Free.

See Also

OOMPI_Comm, OOMPI_Request_array, OOMPI_Status
Name | OOMPI_Request_array  
---|---  
Declaration | #include "oompi.h"  
| class OOMPI_Request_array  
Inheritance | None.  
Description | A class for encapsulating an array of MPI_Request handles and their associated functionality.  
Constructors/Destructors |  
OOMPI_Request_array(int count = 1). Default constructor. An array of MPI_request handles is created of size count.  
OOMPI_Request_array(MPI_Request array[], int count). Constructor. Creates an OOMPI_Request_array object from an array of MPI_Request handles. This operation makes a copy of the given array parameter.  
OOMPI_Request_array(const OOMPI_Request_array& a). Copy constructor. Perform a deep copy of a; OOMPI_Request objects are not reference counted.  
OOMPI_Request_array& operator=(const OOMPI_Request_array& a). Assignment operator. Perform a deep copy of the OOMPI_Request_array object; OOMPI_Request objects are not reference counted.  
~OOMPI_Request_array(). Delete the current memory associated with the current array of requests. Does not trigger a call to MPI_Request_free(). This needs to be handled explicitly by the user (using the Free() or Freeall() methods) if necessary.  
Access and Information |  
OOMPI_Request& operator[](int i). Returns a reference to an OOMPI_Request object that contains the i-th element in the internal MPI_Request array.  
MPI_Request *Get_mpi(void). Returns the internal array of MPI_Request handles.  
void Set_mpi(MPI_Request a[], int size). Sets the underlying MPI_Request request array and size. This function makes a copy of the given array parameter, so it can be re-used by the calling program.  
int Get_size(void). Returns the number of MPI_Request handles in the internal array.  
bool Set_size(int size). Sets the size of the array of MPI_Request handles. Returns a boolean indicating whether the action was successful or not.
bool operator== (const OOMPI_Request_array& a). Returns a bool indicating if a pairwise comparison of every element in the current instance to every element in a returns true, false otherwise.

bool operator!= (const OOMPI_Request_array& a). Returns a bool indicating if a pairwise comparison of every element in the current instance to every element in a returns false, true otherwise.

Test / Wait

OOMPI_Status_array Testall (bool& flag). Calls MPI_Testall() to test if all of the communications associated with the array of MPI requests have completed. flag is set true if all of the operations have completed. An OOMPI_Status_array object is returned that contains an MPI_Status handle for each MPI_Request in the invoking object.

bool Testall (OOMPI_Status_array& status). Shortcut for the previous function, except that it takes status as an argument and returns flag.

OOMPI_Status Testany (int& index, bool& flag). Calls MPI_Testany() to test if any of the operations associated with the array of MPI requests has completed. flag is set true if at least one operation completed and index contains the index of the request that completed. An OOMPI_Status object is returned that contains the MPI_Status of the completed operation. Otherwise, an invalid OOMPI_Status is returned.

bool Testany (OOMPI_Status& status, int& index). Shortcut for the previous function, except that it takes status as an argument and returns flag.

OOMPI_Status_array Testsome (int& outcount, int array_of_indices[]). Calls MPI_Testsome() to test some of the operations associated with the array of MPI requests. outcount is set to the number of operations completed. An OOMPI_Status_array object is returned that contains an MPI_Status handle for each MPI_Request in the invoking object.

int Testsome (OOMPI_Status_array& status, int array_of_indices[]). Shortcut for the previous function, except that it takes status as an argument and returns outcount.

OOMPI_Status_array Waitall (void). Calls MPI_Waitall() to block until all of the operations associated with the array of MPI requests return. An OOMPI_Status_array object is returned that contains an MPI_Status handle for each MPI_Request in the invoking object.

void Waitall (OOMPI_Status_array& status). Shortcut for the previous function, except that it takes the status as an argument, and returns nothing.

OOMPI_Status Waitany (int& index). Calls MPI_Waitany() to block until any one of the operations associated with the array of requests completes. index contains the index of the request that completed. An OOMPI_Status object is returned that contains the MPI_Status of the completed operation.
int Waitany(OOMPI_Status& status). Similar to the above function, except that the index is returned and the status is filled.

OOMPI_Status_array Waitsome(int& outcount, int array_of_indices[]). Calls MPI_Waitsome() to block until at least one of the operations associated with the array of MPI requests completes. It sets outcount to the number of operations completed. An OOMPI_Status_array object is returned that contains an MPI_Status handle for each MPI_Request in the invoking object.

int Waitsome(OOMPI_Status_array& status, int array_of_indices[]). Shortcut for the previous function, except that it takes the OOMPI_Status_array as an argument, and returns outcount.

Start
void Startall(void). Starts all the communications associated with the MPI request array by calling MPI_Startall().

Free
void Freeall(void). Frees (using MPI_Request_Free) all non-null requests in this array. This operation is a no-op on POE 2.1.0.14 (since this version of POE appears to have a bug in MPI_Request_Free).

See Also
OOMPI_Comm, OOMPI_Request, OOMPI_Status
Name: OOMPI_Status

Declaration:
```
#include "oompi.h"
class OOMPI_Status
```

Description:
A class for encapsulating a single MPI_Status handle and its associated functionality.

Inheritance:
None.

Constructors/Destructors:
- `OOMPI_Status(void)`: Default constructor. An invalid instance is created. Since there is no `OOMPI_STATUS_NULL` object, it is not possible to merge the default and MPI constructors into one function.
- `OOMPI_Status(MPI_Status status)`: MPI constructor. A new container is created for `status` and a reference to it is made.
- `OOMPI_Status(const OOMPI_Status& a)`: Copy constructor. Perform a deep copy; OOMPI_Status objects are not reference counted.
- `OOMPI_Status& operator=(const OOMPI_Status& a)`: Assignment operator. Perform a deep copy; OOMPI_Status objects are not reference counted.
- `OOMPI_Status& operator=(const MPI_Status& a)`: Assignment operator. Perform a deep copy; OOMPI_Status objects are not reference counted.
- `~OOMPI_Status()`: Destructor. Delete the current reference to the internal MPI_Status.

Access and Information:
- `int Get_count(OOMPI_Datatype type)`: Calls `MPI_Get_count()` to get the number of entries of datatype that were received.
- `int Get_elements(OOMPI_Datatype type)`: Calls `MPI_Get_elements()` to get the number of received basic elements that were received.
- `int Get_error(void)`: Returns the error code of the message referenced by the status object.
- `MPI_Status& Get_mpi(void)`: Returns a reference to the underlying MPI_Status handle.
- `int Get_source(void)`: Returns the source rank of the message referenced by the status object.
- `int Get_tag(void)`: Returns the tag of the message referenced by the status object.
Test

bool Test_cancelled(). Calls MPI_Test_cancelled() to determine whether the cancel associated the OOMPI_Status object was successful. A true is returned upon a successful cancellation.

See Also

OOMPI_Comm, OOMPI_Datatype, OOMPI_Port, OOMPI_Request
Name       OOMPI_Status_array

Declarati

class OOMPI_Status_array

Description A class for encapsulating an array of MPI_Status handles and their associated functionality.

Inheritance None.

Constructors/Destructors

OOMPI_Status_array(int size = 1). Default constructor. A new container is created for a newly created array of size MPI_Status handles, and a reference to it is made.

OOMPI_Status_array(MPI_Status array[], int size = 1). Constructor. A new container is created for array and a copy of it is made.

OOMPI_Status_array(const OOMPI_Status_array& array). Copy constructor. Performs a deep copy; OOMPI_Status_array objects are not reference counted.

OOMPI_Status_array& operator=(const OOMPI_Status_array& array). Assignment operator. Performs a deep copy; OOMPI_Status_array objects are not reference counted.

~OOMPI_Status_array() Destructor. Frees the memory associated with the internal MPI_Status_array if the memory was allocated by OOMPI.

Access and Information

OOMPI_Status& operator[](int i). Returns a reference to the ith OOMPI_Status in the array.

MPI_Status *Get_mpi(void). Returns a pointer to the underlying MPI_Status status array.

void Set_mpi(MPI_Status a[], int size). Sets the underlying MPI_Status status array and size. This function makes a copy of the array parameter, so it can be re-used by the calling program.

int Get_size(void). Returns the number of MPI_Status handles in the internal array.

bool Set_size(int size). Sets the size of the array of MPI_Status handles. Returns a boolean indicating whether the action was successful or not.

See Also OOMPI_Comm, OOMPI_Datatype, OOMPI_Port, OOMPI_Request
Name

OOMPI_Tag

Declaration

#include "oompi.h"
class OOMPI_Tag

Description

The OOMPI_Tag class is used for inheritance only; it is never explicitly instantiated. Several OOMPI objects inherit from OOMPI_Tag to gain the use of its methods.

Inheritance

None.

Constructors/Destructors

OOMPI_Tag(int tag = OOMPI_NO_TAG). Default constructor. Creates a tag with a sentinel value that, while valid, indicates that no tag has been set.

Access and Information

int Get_tag(void). Returns the value of the tag.

void Set_tag(int tag). Sets the value of the tag.

See Also

OOMPI Constants
Name OOMPI_User_type

Declaration
#include "oompi.h"
class OOMPI_User_type

Description A base class for creating user-defined OOMPI data objects. Classes derived from OOMPI_User_type can immediately use existing OOMPI communication functions, provided the user-defined type is properly constructed.

Inheritance This class inherits functions from OOMPI_Message and OOMPI_Tag.

Constructors/Destructors

OOMPI_User_type(oompi_datatype &type, void *top, int tag). Constructor. Associates an OOMPI datatype with a user object. Its arguments are a pointer to the static OOMPI_Datatype member of the user class (see code example in Section 3.4), this, and the default tag to use for this class. **NOTE:** It is not necessary for the type argument to have been constructed yet; it only needs to be instantiated.

~OOMPI_User_type(). Destructor. Does nothing except internal bookkeeping.

See Also OOMPI_Datatype, OOMPI_Message, OOMPI_Tag
Name: OOMPI Enumerated Types

 Declaration: #include "mpi.h"

 Description: Listed below are OOMPI enumerated types and their possible values.

 OOMPI Enumerated Types

 OOMPI_Aint. This type corresponds to MPI_Aint.

 OOMPI_Compare. This type is returned from the communicator and group Compare() functions.

```
OOMPI_IDENT
OOMPI_CONGRUENT
OOMPI_SIMILAR
OOMPI_UNEQUAL
```

 Default tags. The following are a list of default tags that are used in OOMPI (usually based upon the datatype). Note that all of these values are above OOMPI_TAG_UB, and should never conflict with user tags.

```
OOMPI_CHAR_TAG     OOMPI_SHORT_TAG
OOMPI_INT_TAG      OOMPI_LONG_TAG
OOMPI_UNSIGNED_-
CHAR_TAG          OOMPI_UNSIGNED_SHORT_TAG
OOMPI_UNSIGNED_TAG OOMPI_UNSIGNED_LONG_TAG
OOMPI_FLOAT_TAG   OOMPI_DOUBLE_TAG
OOMPI_BYTE_TAG    OOMPI_MESSAGE_TAG
OOMPI_PACKED_TAG  OOMPI_MPI_DATATYPE_TAG
OOMPI_INTERCOMM_-
CREATE_TAG        OOMPI_NO_TAG
```

 OOMPI_Error_action. This type is used to check and set what OOMPI does when MPI errors are encountered. Valid values are:

```
OOMPI_ERRORS_ARE_FATAL
OOMPI_ERRORS_EXCEPTION
OOMPI_ERRORS_RETURN
```

 LET THE UNDERLYING MPI FUNCTION HANDLE THE ERROR.
 OOMPI COMPLAINS AND THROWS AN OOMPI_ERROR EXCEPTION.
 DO NOTHING. IMPLEMENTATION DEPENDENT ON HOW RELIABLE MPI IS.

 63
OOMPI_Error_type. OOMPI_errno is loaded with a value of this type after an MPI error occurs. The values listed below have the same meanings as their C counterparts.

<table>
<thead>
<tr>
<th>OOMPI_SUCCESS</th>
<th>OOMPI_ERR_TOPOLOGY</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOMPI_ERR_BUFFER</td>
<td>OOMPI_ERR_DIMS</td>
</tr>
<tr>
<td>OOMPI_ERR_COUNT</td>
<td>OOMPI_ERR_ARG</td>
</tr>
<tr>
<td>OOMPI_ERR_TYPE</td>
<td>OOMPI_ERR_UNKNOWN</td>
</tr>
<tr>
<td>OOMPI_ERR_TAG</td>
<td>OOMPI_ERR_TRUNCATE</td>
</tr>
<tr>
<td>OOMPI_ERR_COMM</td>
<td>OOMPI_ERR_OTHER</td>
</tr>
<tr>
<td>OOMPI_ERR_RANK</td>
<td>OOMPI_ERR_INTERN</td>
</tr>
<tr>
<td>OOMPI_ERR_REQUEST</td>
<td>OOMPI_ERR_PENDING</td>
</tr>
<tr>
<td>OOMPI_ERR_ROOT</td>
<td>OOMPI_ERR_IN_STATUS</td>
</tr>
<tr>
<td>OOMPI_ERR_GROUP</td>
<td>OOMPI_ERR_LASTCODE</td>
</tr>
<tr>
<td>OOMPI_ERR_OP</td>
<td></td>
</tr>
</tbody>
</table>

See Also          OOMPI Constants
Name  OOMPI Constants

Declaration  #include "mpi.h"

Description  Listed below are OOMPI global constants and their respective types. They are mainly used for initialization and comparison. It is erroneous to attempt to assign a value to any of these constants.

OOMPI Constants

    int OOMPI_ANY_SOURCE. Has the same meaning as MPI_ANY_SOURCE.

    int OOMPI_ANY_TAG. Has the same meaning as MPI_ANY_TAG.

OOMPI_Comm_world  OOMPI_COMM_WORLD. Singleton instance of the OOMPI_Comm_world class.

Pre-defined datatypes. The following OOMPI_Datatype constants are initialized upon OOMPI_COMM_WORLD::Init():

<table>
<thead>
<tr>
<th>OOMPI_CHAR</th>
<th>OOMPI_SHORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOMPI_INT</td>
<td>OOMPI_LONG</td>
</tr>
<tr>
<td>OOMPI_UNSIGNED_CHAR</td>
<td>OOMPI_UNSIGNED_SHORT</td>
</tr>
<tr>
<td>OOMPI_UNSIGNED</td>
<td>OOMPI_DOUBLE</td>
</tr>
<tr>
<td>OOMPI_FLOAT</td>
<td>OOMPI_UNSIGNED_LONG</td>
</tr>
<tr>
<td>OOMPI_BYTE</td>
<td>OOMPI_MESSAGE</td>
</tr>
<tr>
<td>OOMPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

OOMPI_Error_action  OOMPI_errno. Contains the error code of the last OOMPI function called (which will be OOMPI_SUCCESS) until an error occurs. This variable is *not* reset back to OOMPI_SUCCESS after an error has occurred; the MPI standard states that implementations do not have to guarantee the stability of the internal state of MPI after an error. Therefore, after an MPI error, OOMPI is only as reliable as the underlying MPI implementation, and OOMPI_errno reflects this potential instability.

OOMPI_Environment  OOMPI_ENV. Singleton instance of the OOMPI_Environment class.

OOMPI_Group  OOMPI_GROUP_EMPTY  OOMPI version of MPI_GROUP_EMPTY.

OOMPI_Group  OOMPI_GROUP_NULL  OOMPI version of MPI_GROUP_NULL.

    int OOMPI_HOST. This integer is initialized in OOMPI_COMM_WORLD::Init(). It corresponds to the integer attribute that the MPI_HOST keyval can be used to retrieve.

    int OOMPI_IO. This integer is initialized upon OOMPI_COMM_WORLD::Init(). It corresponds to the integer attribute that the MPI_IO keyval can be used to retrieve.

Pre-defined operations. The following OOMPI_Op constants are initialized in OOMPI_COMM_WORLD::Init():
<table>
<thead>
<tr>
<th>OOMPI_MAX</th>
<th>OOMPI_MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOMPI_SUM</td>
<td>OOMPI_PROD</td>
</tr>
<tr>
<td>OOMPI_MINLOC</td>
<td>OOMPI_MAXLOC</td>
</tr>
<tr>
<td>OOMPI_BAND</td>
<td>OOMPI_BOR</td>
</tr>
<tr>
<td>OOMPI_BXOR</td>
<td>OOMPI_LAND</td>
</tr>
<tr>
<td>OOMPI_LOR</td>
<td>OOMPI_LXOR</td>
</tr>
</tbody>
</table>

OOMPI Port OOMPI_PORT_NULL. This port is analogous to MPI_PROC_NULL; any communications on it will immediately return.

int OOMPI_PROC_NULL. Has the same meaning as MPI_PROC_NULL.

int OOMPI_RESERVED_TAGS. The number of tags that OOMPI reserves for internal use.

int OOMPI_TAG UB. This integer is initialized in OOMPI_COMM_WORLD::Init(). It corresponds to the integer attribute that the MPI_TAG UB keyval can be used to retrieve. However, since OOMPI reserves the upper OOMPI_RESERVED_TAGS tags, OOMPI_TAG UB actually equals the MPI implementation’s upper bound on tags minus OOMPI_RESERVED_TAGS.

int OOMPI_UNDEFINED. Has the same meaning as MPI_UNDEFINED.

bool OOMPI_WTIME_IS_GLOBAL. This bool is initialized in OOMPI_COMM_WORLD::Init(). It corresponds to the integer attribute that the MPI_WTIME ISLOBAL keyval can be used to retrieve.

See Also

OOMPI_Comm_world, OOMPI_Datatype, OOMPI_Environment, OOMPI_Op, OOMPI_Packed, OOMPI_Port
6 Availability

Version 1.0.4 of an open-source implementation of OOMPI can be found at the URL listed below. This version has been implemented as a thin layer on top of the MPI C bindings; it can be installed on top of any MPI-1.1 conformant implementation.

http://www.osl.iu.edu/research/oompi/

This WWW site will always contain the latest information about releases, documentation, patches, etc.

6.1 The OOMPI Distribution

Version 1.0.4 of the OOMPI distribution includes a full implementation of all the classes and member functions listed in Section 5.1, several example programs (including the ring programs from Sections 4.1 and 4.2), and a complete OOMPI verification suite. The verification suite tests every class and member function with the C++ compiler and underlying MPI implementation.

6.2 Contact Information

If you use OOMPI, we would like to know. This will help us in understanding how large the OOMPI user community is. As graduate students, we like to envision that there is some “real world” outside of our office, and someday we hope to see it. Can you help us out? Send us a snail mail picture postcard of your area in this “real world” that we’ve heard so much about:

OOMPI / Open Systems Lab
Department of Computer Science
Lindley Hall 215
150 S. Woodlawn Ave.
Bloomington, IN 47405-7104

Your postcard will be scanned in and displayed on the OOMPI WWW site.
A broadcast mailing list exists for OOMPI announcements regarding releases, important patches, etc. Users can subscribe by visiting the OOMPI web site.
Questions, comments, and bug reports can be directed to the general OOMPI user’s mailing list. In order to control spam, you must be a member of the list in order to post to the list. Visit the OOMPI WWW to subscribe to the ompi-devel mailing list.

oompi-devel@osl.iu.edu
Index

MPI_Abort, 25
MPI_Address, 30, 31
MPI_Allgather, 41
MPI_Allgatherv, 41
MPI_Allreduce, 41
MPI_Alltoall, 41
MPI_Alltoallv, 41
MPI_Attr_Get, 10
MPI_Attr_Put, 10
MPI_Barrier, 41
MPI_Bcast, 50
MPI_Bsend, 50
MPI_Bsend_init, 50
MPI_Buffer_attach, 32
MPI_Cancel, 54
MPI_Cart_coords, 22
MPI_Cart_create, 21
MPI_Cart_get, 22
MPI_Cart_Map, 42
MPI_Cart_rank, 22
MPI_Cart_shift, 23
MPI_Cart_sub, 21
MPI_Cartdim_get, 22
MPI_Comm_compare, 25
MPI_Comm_create, 41
MPI_Comm_dup, 21, 34, 39, 41
MPI_Comm_free, 24, 49
MPI_Comm_rank, 25, 49
MPI_Comm_remote_group, 39
MPI_Comm_remote_size, 39
MPI_Comm_size, 25
MPI_Comm_split, 41
MPI_Comm_test_inter, 25
MPI_Dims_create, 42
MPI_Finalize, 26, 27
MPI_Gather, 50
MPI_Gatherv, 50
MPI_Get_count, 58
MPI_Get_elements, 58
MPI_Get_processor_name, 32
MPI_Graph_create, 34
MPI_Graph_get, 34, 35
MPI_Graph_Map, 42
MPI_Graph_neighbors, 35
MPI_Graph_neighbors_count, 35
MPI_Graphdims_get, 34
MPI_Group, 25
MPI_Group_compare, 37
MPI_Group_difference, 37
MPI_GROUP_EMPTY, 36
MPI_Group_excl, 38
MPI_Group_free, 36
MPI_Group_incl, 38
MPI_Group_intersection, 37
MPI_Group_range_excl, 38
MPI_Group_range_incl, 38
MPI_Group_rank, 36
MPI_Group_size, 36
MPI_Group_translate_ranks, 36
MPI_Group_union, 37
MPI_Ibsend, 51
MPI_Init, 26
MPI_Initialized, 24
MPI_Intercomm_create, 39, 49
MPI_Intercomm_merge, 39
MPI_Iprobe, 42
MPI_Irecv, 51
MPI_Irsend, 51
MPI_Isend, 51
MPI_Keyval_create, 10
MPI_Keyval_free, 10
MPI_LB, 30
MPI_Op_create, 45
MPI_Op_free, 46
MPI_Pack, 48
MPI_Pack_size, 24, 25, 47
MPI_Pcontrol, 32
MPI_Probe, 42
MPI_Recv, 42, 50, 51
MPI_Recv_init, 52
MPI_Reduce, 50
MPI_Reduce_scatter, 41
MPI_Request_Free, 54, 57
MPI_Request_free, 53, 55
MPI_Rsend, 51
MPI_Rsend_init, 51
MPI_Scan, 41
MPI_Scatter, 50
MPI_Scatterv, 50
MPI_Send, 50, 51
MPI_Send_init, 51
MPI_Sendrecv, 25
MPI_Sendrecv_replace(), 25
MPI_Ssend, 51
MPI_Ssend_init, 51
MPI_Start, 54
MPI_Startall, 57
MPI_Test, 53
MPI_Test_cancelled, 59
MPI_Testall, 56
MPI_Testany, 56
MPI_Testsome, 56
MPI_Type_commit, 29–31
MPI_Type_contiguous, 29
MPI_Type_extent, 29
MPI_Type_free, 28
MPI_Type_hindexed, 29, 30
MPI_Type_hvector, 30
MPI_Type_indexed, 30
MPI_Type_lb, 29
MPI_Type_size, 29
MPI_Type_struct, 31
MPI_Type_ub, 29
MPI_Type_vector, 30
MPI_Unpack, 48
MPI_Wait, 54
MPI_Waitall, 56
MPI_Waitany, 56
MPI_Waitsome, 57
MPI_Wtick, 32
MPI_Wtime, 32
MPI_Reduce, 45
MPI_Reduce_scatter, 45
MPI_Type_commit, 9
MPI_Type_struct, 9
OOMPI_Aint, 63
OOMPI_ANY_SOURCE, 65
OOMPI_ANY_TAG, 65
OOMPI_Array_message, 19
OOMPI_BAND, 66
OOMPI_BOR, 66
OOMPI_BXOR, 66
OOMPI_BYTE, 65
OOMPI_BYTE_TAG, 63
OOMPI_Cart_comm, 21
OOMPI_CHAR, 65
OOMPI_CHAR_TAG, 63
OOMPI_Comm, 9, 24, 39
OOMPI_COMM_WORLD, 24, 26, 33, 65
OOMPI_Comm_world, 26
OOMPI_CONGRUENT, 63
OOMPI_Datatype, 28
OOMPI_DOUBLE, 65
OOMPI_DOUBLE_TAG, 63
OOMPI_ENV, 65
OOMPI_Environment, 32
OOMPI_errno, 9
OOMPI_Error, 33
OOMPI_Error_action, 63, 65
OOMPI_Error_type, 64
OOMPI_ERRORS_ARE_FATAL, 24, 63
OOMPI_ERRORS_EXCEPTION, 63
OOMPI_ERRORS_RETURN, 63
OOMPI_FLOAT, 65
OOMPI_FLOAT_TAG, 63
OOMPI_Graph_comm, 34
OOMPI_Group, 36
OOMPI_GROUP_EMPTY, 65
OOMPI_GROUP_NULL, 65
OOMPI_HOST, 65
OOMPI_IDEENT, 63
OOMPI_INT, 65
OOMPI_INT_TAG, 63
OOMPI_Inter_comm, 39
OOMPI_INTERCOMM_CREATE_TAG, 63
OOMPI_Intra_comm, 21, 26, 40
OOMPI_IO, 65
OOMPI_LAND, 66
OOMPI_LONG, 65
OOMPI_LONG_TAG, 63
OOMPI_LOR, 66
OOMPI_LXOR, 66
OOMPI_MAX, 66
OOMPI_MAXLOC, 66
OOMPI_MESSAGE, 65
OOMPI_Message, 43, 62
OOMPI_MESSAGE_TAG, 63
OOMPI_MIN, 66
OOMPI_MINLOC, 66
OOMPI_MPI_DATATYPE_TAG, 63
OOMPI_NO_TAG, 63
OOMPIOp, 45
OOMPI_PACKED, 65
OOMPI_Packed, 18, 47
OOMPI_PACKED_TAG, 63
OOMPI_Port, 49
OOMPI_PORT_NULL, 66
OOMPI_PROC_NULL, 66
OOMPI_PROD, 66
OOMPI_Request, 18, 53
OOMPI_Request_array, 18, 55
OOMPI_RESERVED_TAGS, 7, 66
OOMPI_SHORT, 65
OOMPI_SHORT_TAG, 63
OOMPI_SIMILAR, 63
OOMPI_Status, 18, 58, 60
OOMPI_Status_array, 18
OOMPI_SUM, 66
OOMPI_Tag, 28, 43, 47, 61, 62
OOMPI_TAG UB, 7, 63, 66
OOMPI_UNDEFINED, 9, 66
OOMPI_UNEQUAL, 63
OOMPI_UNSIGNED, 65
OOMPI_UNSIGNED_CHAR, 65
OOMPI_UNSIGNED_CHAR_TAG, 63
OOMPI_UNSIGNED_LONG, 65
OOMPI_UNSIGNED_LONG_TAG, 63
OOMPI_UNSIGNED_SHORT, 65
OOMPI_UNSIGNED_SHORT_TAG, 63
OOMPI_UNSIGNED_TAG, 63
OOMPI_User_type, 62
OOMPI_WTIME_IS_GLOBAL, 66
References


