Object-Oriented Analysis and Design of the Message Passing Interface*

Anthony Skjellum† Diane G. Wooley
Ziyang Lu Michael Wolf‡
Purushotham V. Bangalore
Engineering Research Center &
Department of Computer Science
Mississippi State University
Mississippi State, MS 39762
{tony,dwooley}@cs.msstate.edu
puri@erc.msstate.edu

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

July 28, 1998

Abstract

The major contribution of this paper is the application of modern analysis techniques to the important Message Passing Interface standard, work done in order to obtain information useful in designing both application programmer interfaces for object-oriented languages, and message passing systems. Recognition of "Design Patterns" within MPI is an important discernment of this work. A further contribution is a comparative discussion of the design and evolution of three actual object-oriented designs for the Message Passing Interface (MPI-1) application programmer interface (API), two of which have influenced the standardization of C++ explicit parallel programming with MPI-2, and which strongly indicate the value of a priori object-oriented design and analysis of such APIs. Knowledge of design patterns is assumed herein.

Discussion provided here includes systems developed at Mississippi State University (MPI++), the University of Notre Dame (OOMPI), and the merger of these systems that results in a standard binding within the MPI-2 standard. Commentary concerning additional opportunities for further object-oriented analysis and design of message passing systems and APIs, such as MPI-2 and MPI/RT are mentioned in conclusion.

Connection of modern software design and engineering principles to High Performance Computing programming approaches is a new and important further contribution of this work.

*Work at the University of Notre Dame was supported by National Science Foundation cooperative grants CCR94-22380 and CCR94-22381. Support for work at Mississippi State University supported by the Defense Advanced Research Projects Agency (DARPA) through the U. S. Air Force Research Laboratory under contract F30602-96-1-0329, and with additional support from the National Science Foundation, Early Career Program, ASC-95-01917. Submitted to *Concurrency: Practice & Experience,* July, 1998.

†Corresponding author: +1-601-325-8435, FAX: +1-601-325-8997.

‡Present address, Stanford Linear Accelerator, Stanford, CA.

1 Introduction

The message passing paradigm for parallel computing is widely used and well understood. Message passing communication layers are commonly used on many types of machines, ranging from massively parallel supercomputers to clusters of workstations. The reader is assumed here to have a significant knowledge of the de facto MPI-1 standard [14, 15], an effort influenced by over a decade of research and commercial systems [2, 3, 7, 11, 12, 19, 21, 23, 24, 25, 27, 29]. This interface has been realized in several commercial implementations, based on well-known public implementations [5, 20].

In this paper, we present two object-oriented class libraries for supplementing MPI, in support of C++. The features and limitations of these systems are indicated, as is an after-the-fact object-oriented analysis of the application programmer interface and semantics of MPI-1. We compare the two class libraries to the C++ interface accepted by the MPI Forum. Furthermore, we note that both class libraries have formed the basis for support of this interface in various forms of MPI implementations. The main lessons here are that C++ needed a class library to work comfortably with the MPI application programmer interface, that several levels of abstraction make sense, and that these still may be particularly of interest to users in view of the spartan support ultimately introduced by the MPI Forum itself.

The major contribution of this paper is the application of modern analysis techniques to the important Message Passing Interface standard, work done in order to obtain information useful in designing both application programmer interfaces for object oriented languages, and message passing systems. Recognition of "Design Patterns" within MPI is an important discernment of this work. A further contribution is a comparative discussion of the design and evolution of three actual object-oriented designs for the Message Passing Interface (MPI-1) application programmer interface (API), two of which have influenced the standardization of C++ explicit parallel programming with MPI-2, and which strongly indicate the value of a priori object-oriented design and analysis of such APIs. Knowledge of design patterns is assumed herein.

Discussion provided here includes systems developed at Mississippi State University (MPI++), the University of Notre Dame (OOMPI), and the merger of these systems that results in a standard binding within the MPI-2 standard. Commentary concerning additional opportunities for further object-oriented analysis and design of message passing systems and APIs, such as MPI-2 and MPI/RT are mentioned in conclusion.

Connection of modern software design and engineering principles to High Performance Computing programming is a timely and important further contribution of this work.

1.1 C++ and MPI

MPI-1 included bindings for the C and Fortran77 languages. Since C++ bindings were not included in the MPI-1 standard, C++ programmers were obliged to augment the MPI C binding in order to achieve feasible programming with MPI, and this factor most probably detracted from use of MPI by C++ programmers in the mid-1990's.

Two research groups, reporting jointly in this communication, prepared class libraries with distinct design factors and features, in order to address this situation: MPI++, designed at Mississippi State University, and Object-Oriented MPI (OOMPI), designed by the University of Notre Dame. Both MPI++ and OOMPI were explicitly designed for inheritance; users can derive their own objects from either class library. MPI++ uses several features of C++ that are absent from the subsequently accepted MPI-2 C++ bindings, while remaining faithful to the majority of function signatures from the MPI C bindings. OOMPI emphasizes object-oriented design and ease of use rather than compliance with the MPI C bindings. In addition to reference and `const` semantics, OOMPI makes extensive use of default arguments, overloaded function names, and inheritance. Both libraries are of interest because they preceeded C++ bindings that were later added to MPI and significantly influenced these standardized extensions.

In particular, the MPI C++ bindings were accepted to be a minimalistic approach to the message passing model. A small set of objects are provided which encapsulate all MPI data and functionality. While much of the C function signatures are preserved, the C++ bindings take advantage of several inherent features of the C++ language, to include reference and `const` semantics. More advanced features of C++, such as overloading and polymorphism, were not used in order preserve a direct and unambiguous mapping to the specified functionality of MPI. While the design criteria appear restrictive, the MPI C++ bindings provide both a simple object-oriented model that programmers can immediately use in their C++ programs as well as a sound basis for building class libraries that use more advanced features of C++.

1.2 Related Work

In this section, we note other efforts to enhance MPI and other message passing systems for use with C++.

1.2.1 ARCH

ARCH is a C++-based library for asynchronous and loosely synchronous system programming, particularly targeted at dynamic, irregular problems [1]. ARCH provides support for this type of programming with facilities such as dynamic creation of threads and processes as well as load balancing strategies. Although ARCH is designed on top of the MPI interface and supports calls to the MPI functions, its communication interface is vastly different from MPI's. For example, channels for communications between processes must be defined and the processes must be synchronized.

1.2.2 mpi++

The mpi++ library (not to be confused with MPI++) is a C++ binding for MPI that is supposed to present the semantic and conceptual model of MPI in a way which is easily recognizable to those familiar with MPI and is also a "good" C++ design [22]. mpi++'s concentration on the separation of the message into static and dynamic properties led to a layered construction of the message using templates. Although this method hides many of the details from the user, all information about the message must be known at compile time. The mpi++ package is therefore unable to support data-dependent messages.

1.2.3 Para++

Para++ [9, 10]. provides a C++ binding to use with either PVM [18], or MPI and is designed so that it can be used without a knowledge of either underlying middleware. The main contribution of Para++ is to provide new `io-streams` that allow for communication between processes. Para++ provides the two addition streams `pin` (parallel in) and `pout` (parallel out) that are similar to `cin` and `cout` except they communicate between processes.

1.3 Organization

The remainder of the paper is organized as follows. An object-oriented analysis of MPI is presented in Section 2. Three object-oriented designs for MPI are presented in Sections 3- 5. The three designs are discussed in Section 6. Finally, suggestions for future work and conclusions are offered in Section 7- 8 and 9, respectively.

2 Object-Oriented Analysis of MPI

An object-oriented programming model would contribute to the understanding of the application programmer interface, offer intuition for the design of object-oriented support for such programming, and motivate strategies for extensions that provided added functionality and/or higher achievable performance. The existing C binding for MPI is not entirely suitable for object-oriented programming in C++. For example, the "handles" provided to the user are typically behaving as plain C pointers, so there is no direct mechanism for inheritance and the user cannot customize the environment to better suit a particular application.

The first step in creating object-oriented support for MPI is to derive an object-oriented analysis of the existing MPI bindings. The MPI specification is evidently object based. The persistent opaque objects defined by MPI, such as communicators and groups, fit naturally into an object-oriented programming paradigm. However, the computational models supported by MPI can be better supported if these opaque objects are real objects instead of handles. First, an informal object model is constructed to lay out the overall structure according to the MPI specification. Then, a more detailed design is obtained by inspecting the Argonne/MSU MPICH implementation [20] and using design patterns [17]. A short set of illustrative requirements are posed, together with solutions for such requirements. As C++ is the intended language for implementation, certain design decisions are slanted toward C++. Other object-oriented languages could also be considered in future studies, but are beyond the scope of this paper.

2.1 Developing the MPI Object Model

The Object Model is developed following the steps suggested in the Object Modeling Technique (OMT)[1]:

1. *Studying the* MPI *standard and identifying all the important classes.* Since the MPI specification is informally object-based, many of the classes are readily inferrable. A class corresponds to each opaque object defined in MPI. These classes include: `Communicator`, `Group`, `Datatype`, `Op`, `Request`, `Status`, and `Errorhandler` [14]. Corresponding to the concept of different types of communicators, there is an inferrable class `InterComm` for intercommunicators and `IntraComm` for intracommunicators. Datatypes are also intuitively distinguishable: `BasicType` for predefined datatypes, and `DerivedType` for user derived datatypes. The `Cache` class deals with the "caching" facility specified by MPI. From process topologies, two more classes are identified: `Graph` for general graph virtual topologies, and `Cart` for Cartesian topologies, themselves evidently subclasses of communicators.

   There are three more fundamental classes that are not obvious from the MPI specification: `Message`, `Transports` and `Device`. The word *message* appears many times in the MPI standard. However, since MPI for the most part is a functional specification, it always specifies the exact syntax with parameters when describing sending and receiving, so it is easy to overlook this class. MPI also specifies different types of send and receive operations, `Transports` provides a method of encapsulating these different operations. Similarly, the MPI specification uses different terms, such as "underlying mechanism," "communication system," and "machines" in order to describe the support for passing messages, but the concept of having an abstract `Device` is not discussed.
2. *Preparing a data dictionary.* Table 1 shows a data dictionary for these classes. This is a simple data dictionary, describing only the most important properties of each class.
3. *Identifying associations between classes.* Again, this task was greatly aided by the detailed specification given by the MPI standard itself. The identified associations are illustrated in the object model shown in Figure 1.
4. *Identifying attributes.* The MPI standard specifies explicitly what the opaque objects are, it is not hard to find the critical attributes. For example, communicators must have *context*, but that context is not necessarily a tangible quantity or object. The identified attributes are illustrated in the object model shown in Figure 1.
5. *Refining with inheritance.* Most of the inheritance relationships between MPI classes are clear. For instance, the MPI standard says that topology structures are intracommunicators. The MPI datatypes also form a natural inheritance relationship.

2.2 Separating the Interface and Implementation

The Object Model in Figure 1 captures the structure of the core of MPI. However, the model does not explicitly address how to support the user interface (the 128 functions specified in the MPI-1.1 standard). The specific format of these functions depends on the language bindings. The C binding, which can be specified as a thin layer on top of this design, is not discussed here. Instead, we are interested in an object-oriented design of MPI, so the user interface discussed here will be based on the draft C++ binding of MPI-2 [15], which in turn was influenced by our earlier work.

To support this user interface in detail, a critical issue needs first to be solved. According to the MPI Object Model we have assembled, there are classes corresponding to MPI opaque objects (*e.g.*, communicators) in the underlying system. The user interface also defines similar classes. There are two choices: either

[1]The study presented here considered only MPI-1.

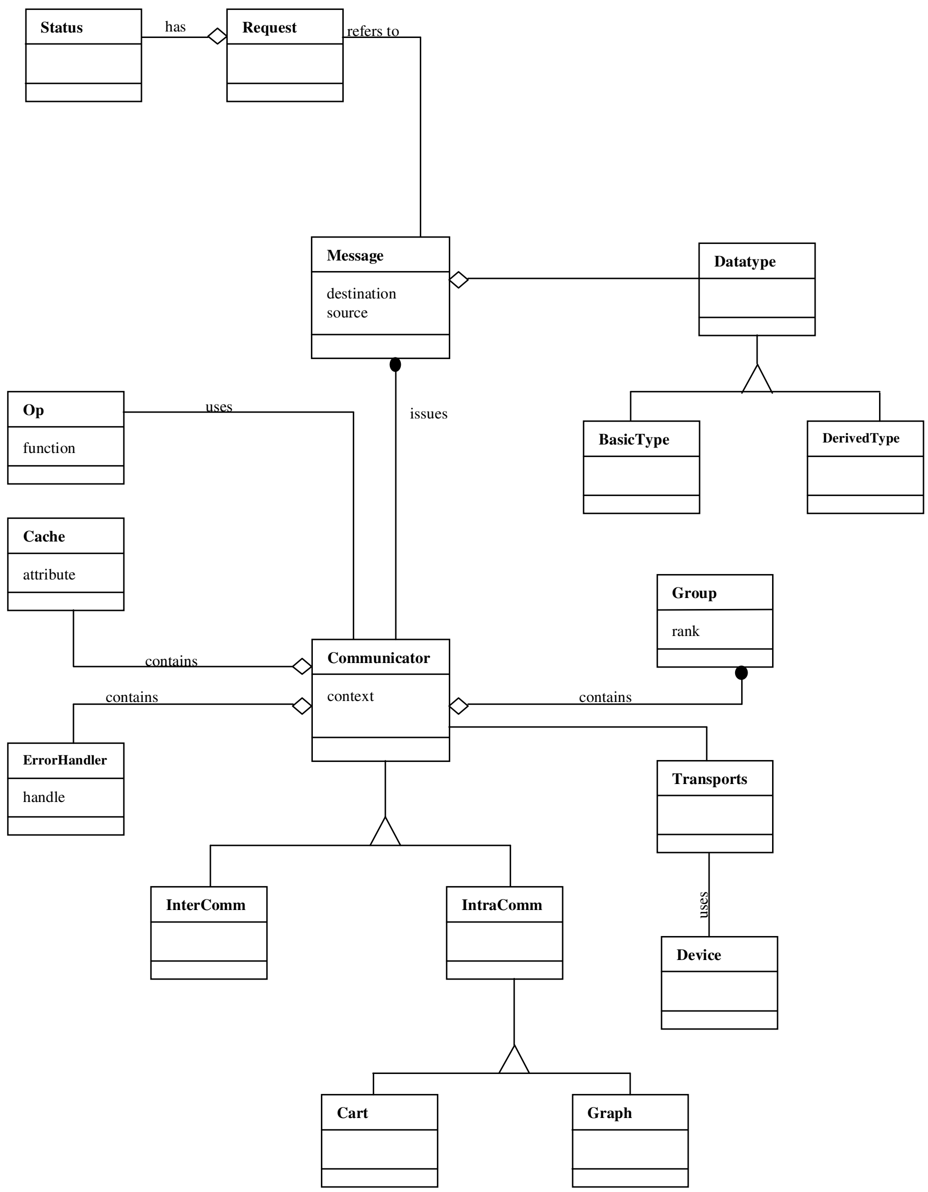

Figure 1: High Level Architectural Model for MPI-1

Class	Description
`Group`	An ordered collection of processes
`Communicator`	The holder of the Group and other information
`IntraComm`	A communicator working on a single group
`InterComm`	A communicator working between two groups
`Cart`	An intracommunicator with Cartesian topology
`Graph`	An intracommunicator with general graph topology
`Datatype`	A type describing the kind of data being transferred
`BasicType`	Predefined datatypes
`Derivedtype`	User-defined datatypes
`Message`	An object that facilitates the composition of data
`Transports`	An object that facilitates the mode of transfer
`Device`	An object that facilitates the transfer of data
`Request`	A handle to inquire about messages
`Status`	The holder of information when a transfer completes
`Cache`	An object that contains attributes, with callbacks
`ErrorHandler`	A handler for processing errors

Table 1: Data Dictionary for MPI-1 Classes.

eliminate the corresponding classes in the Object Model and define the functionality in the user classes, or retain all the classes and treat them differently.

The better approach is the latter. Even though these classes both correspond to MPI opaque objects, the classes in the user interface correspond to handles in the C language binding. The classes in the Object Model are truly used to implement the MPI opaque objects. More specifically, the need for the separation of interface from implementation stems from the following considerations:

1. The MPI standard specifies that MPI manages system memory such as internal representations of various MPI objects like groups, communicators, and datatypes. These structures are not directly accessible to the user, and objects stored there are opaque: their size, shape, and contents are not directly visible to the user. Opaque objects are accessed via handles, which exist in user space [14]. Therefore, in the object-oriented design, there is a need to separate objects into two categories: user objects and system objects.
2. Because of the principle of "design for inheritance" [8], which is to promote flexibility, the draft C++ binding proposes to declare user level methods virtual. This practice would allow the user to refine the methods to suit a particular application better. But this also adds problems in the design and implementation. For example, MPI collective operations are usually implemented at present with point-to-point operations. So, if user level operations are not totally isolated from the implementation, a refined user method (say a point-to-point send operation) may cause unexpected problems on other operations (say the broadcast operation) [14].

The requirement of completely hiding the implementation from the user poses a particular problem if C++ is the implementation language. In C++, even if implementation details are declared as private parts of a class, the user can still see them in the header files.

The Bridge design pattern [17] can be used to set up the structure and solve these problems. According to the Bridge pattern, there should be two sets of classes, one for the user interface, and one purely for implementation. The user object maintains a reference to an object of an implementation class, and forwards client requests to the implementation object. We will use communicator-based classes to illustrate the idea.

Figure 2 shows the Bridge pattern structure for communicator-based classes. `MPI::Comm` has a reference to the abstract base class `Communicator`. The reference will be set to an appropriate concrete subclass of `Communicator` through communicator constructors. The methods of `MPI::Comm` simply forward the request through its reference. The following C++ pseudo-code is meant to clarify this description:

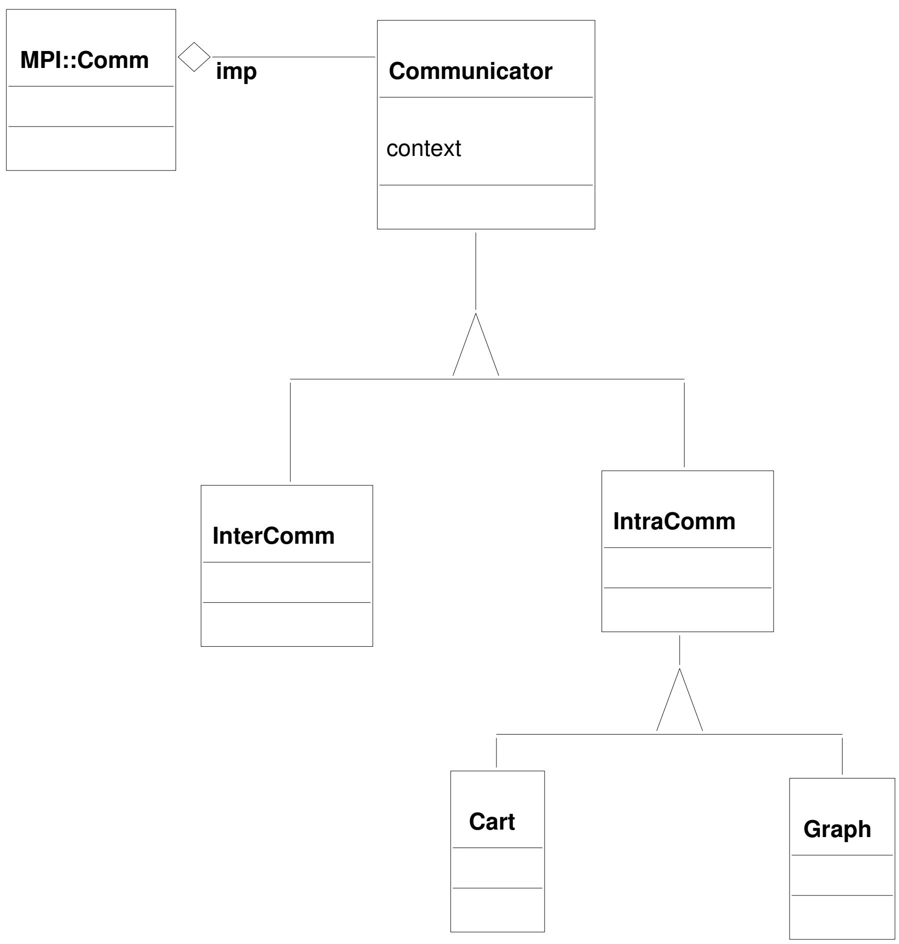

Figure 2: The Bridge Pattern Applied to Communicator-Based Classes in MPI-1

```
class MPI::Comm {
  public:
    // Communicator constructors
    MPI::Comm() : _imp( 0 ) {}
    virtual MPI::~Comm();
    virtual int Create( const MPI::Comm& comm, const MPI::Group& group );
    ...
    // Methods
    virtual int Size ( int& size ) const;
    ...
  private:
    Communicator *_imp;  // the reference through which
                         // implementation is accessed
};
int MPI::Comm::Create( const MPI::Comm& comm, const MPI::Group& group )
{
  _imp = new IntraComm( comm, group );
}
int MPI::Comm::Size( int& size ) const
{
  return _imp->Size( size );
}
```

The Bridge pattern provides a solution to support an object-oriented C++ user interface for MPI without violating the MPI specification. Moreover, the decoupling of interface and implementation allows for changing the underlying implementation freely without affecting the MPI C++ binding; therefore, user programs remain unchanged under such potential perturbations.

2.3 Designing Transports as a Command

The above MPI Object Model recognizes these key concepts *message*, *communicator*, *transports*, and *device* and includes each as a class. Basically, `Communicator` issues a request to send a message and `Transports` is responsible for using the proper send or receive operation and `Device` is responsible for actually getting the message to its destination. The question is how to do that in a flexible and elegant way. To simplify participating classes, ideally, the `Communicator` class should know nothing about how `Transports` actually sends a message, but it must clearly interact with `Transports`. MPI specifies different kinds of send and receive operations [14], it is desirable to be able to handle different messages conveniently, while preserving MPI semantics. Likewise, `Device` abstracts into simple mux/demux information aspects of the message, but is not intimately tied to `Transports`.

The Command design pattern provides a solution by providing a means of encapsulating these different types of send/receive operations. Figure 3 shows the structure obtained by applying the Command pattern to this case. The base class `Transports` declares an interface for executing the `start()` operation, which is to start sending or receiving messages. A concrete subclass of `Transports` uses proper `Device` operations to implement its start() operation. The `Communicator` class knows which subclass to use to instantiate a message object and which sublcass to use for the transports object, but has no idea how the message is transferred. The `Transports` provides message-passing operations and knows nothing about the `Communicator`'s structure, though it will need data placed in `Message` from `Communicator` indirectly, particularly for mux/demux associated with a specific communicator's context space, and the other aspects of the message envelope needed to route and demultiplex.

The Command design pattern sets up the structure, but there are some important details that are not addressed. For example, MPI specifies that messages are non-overtaking: if a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending [14]. To guarantee this partial ordering in the implementation,

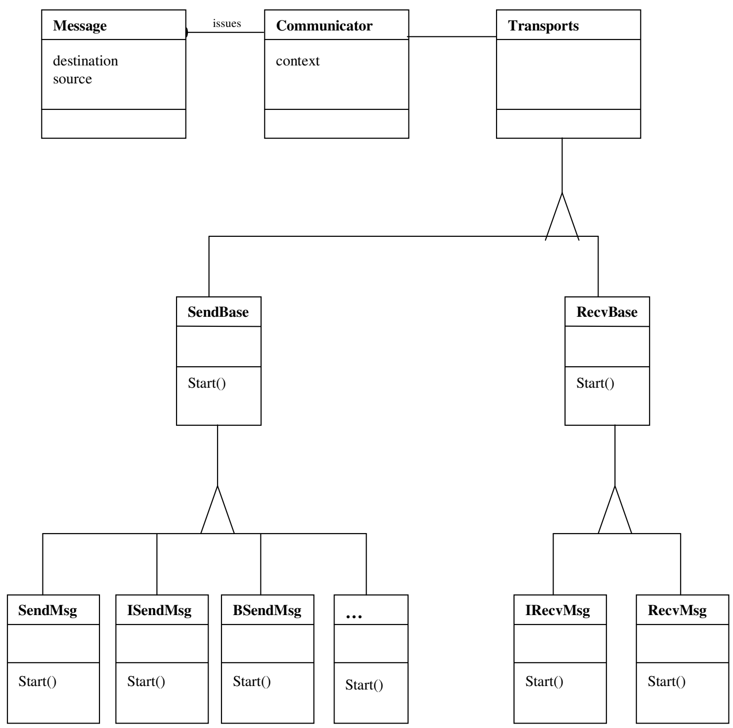

Figure 3: The Command Pattern Applied to Transports in MPI-1

it is necessary to use communicator-based information (group, context information) as well as source and tag information, so the `Message` class has to be carefully designed.

The use of the Command pattern in this case is a little different from the typical situation described in [17] that the receiver in our case is always `Device`, while in general different commands have different receivers. As a matter of fact, explicitly specifying the receiver is not necessary in this case, as we will see that the Device[2] object will follow the Singleton pattern.

2.4 Ensuring Uniform Treatment of Datatypes

MPI has strong support for datatypes. In addition to the basic datatypes such as `int`, `float`, and `char`, MPI provides functions for building derived datatypes, including contiguous, vector, hvector, indexed, hindexed, and structured types [14]. New datatypes are built recursively from previously defined datatypes. Moreover, basic datatypes and newly defined datatypes are used the same way in communication [14].

In the MPI Object Model we have defined, datatypes have been arranged in an inheritance hierarchy. This ensures that all the datatypes can be used the same way as the base class. But the problem of how to build new datatypes recursively from previously defined datatypes has not been addressed. To solve this problem in a clean and elegant way, the Composite design pattern can be used.

The Composite pattern arranges objects into a part-whole hierarchy so that the difference between compositions of objects and individual objects can be ignored from the user's point of view. It allows complex objects to be built from primitive objects, which in turn can be composed, and so on recursively [17]. This matches exactly what is needed here.

Figure 4 shows the structure obtained by applying the Composite pattern to datatypes. Here the aggregation link from the derived datatypes means that the previously defined datatypes building this datatype need to be stored, among other things. The operations of a derived datatype are typically implemented in terms of operations of its building datatypes, with possibly additional actions.

2.5 Abstracting from Specific Devices

The class `Device` in the MPI Object Model represents the underlying communication system (e.g., socket API for TCP/IP or a portable abstraction of packet transfer). While it is sufficient to use this one class to capture the idea in the Object Model, it is rarely so in a real portable implementation. The problem is that the underlying hardware platforms vary greatly. Some architectures use a shared-memory programming model, while others use a distributed-memory model. Even for similar platforms, such as distributed-memory parallel machines, different vendors provide different, incompatible, native communication libraries. So it is impossible to encapsulate all these differences in one class in a practical implementation.

The usual object-oriented solution to this problem would be to treat `Device` as an abstract base class that defines the common interface for handling messages, and derive, by inheritance, subclasses with specific implementations for different platforms. In a sense, this is similar to the Adapter pattern, where a intermediate class is introduced to adapt the interfaces of incompatible existing classes to a common, desired interface [17]. In our case, the existing incompatible mechanisms are in the form of primitive routines instead of classes.

There is one remaining problem: how do the Transports access the Device? One solution is to pass a reference to messages when they are created. There should exist only one instance of a particular device in the system. Therefore a reasonable solution is to make the device well known and guaranteed to be unique.

The Singleton pattern does precisely this, and the structure resulted from applying the Singleton pattern is shown in Figure 5. The following C++ pseudo-code illustrates how this can be achieved:

```
class ConcreteDevice : public Device {
  public:
    // The sole operation to access device
    static ConcreteDevice* instance();
```

[2]Naming follows the common convention of referring to transports interfaces of popular MPI implementations as "devices".

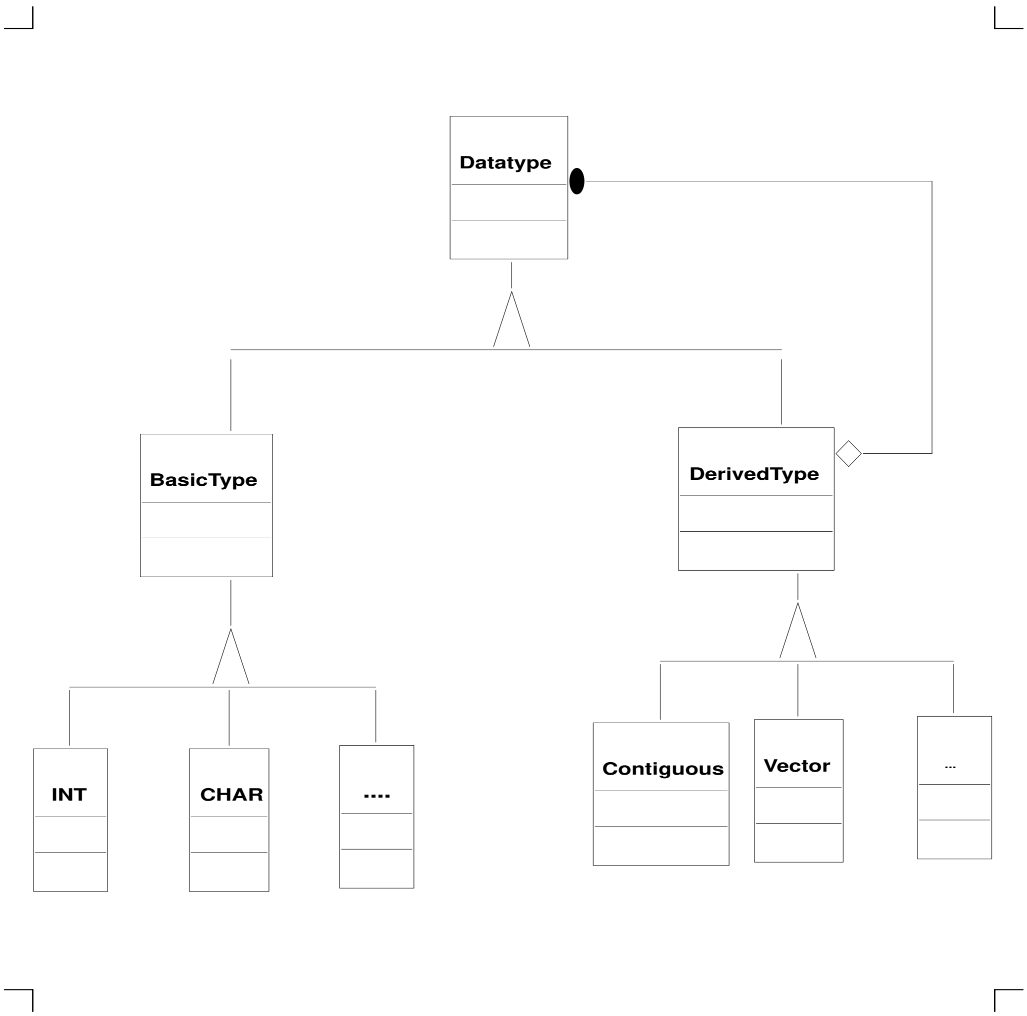

Figure 4: The Composite Pattern Applied to Datatypes in MPI-1

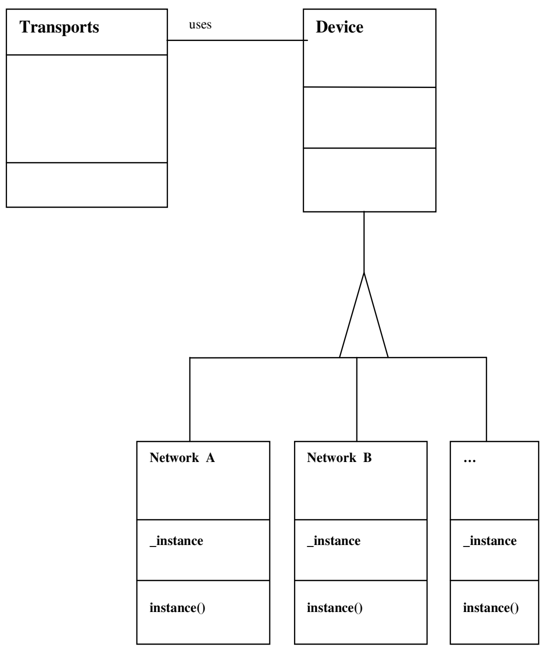

Figure 5: The Singleton Pattern Applied to Devices in MPI-1

```
    // Regular methods
    int deliver();
    ...
  private:
    // Direct instantiation is forbidden
    ConcreteDevice();
    // The only instance
    static ConcreteDevice *_instance;
};
// Static initialization
ConcreteDevice* ConcreteDevice::_instance = 0;
ConcreteDevice* ConcreteDevice::instance()
{
  if ( _instance == 0 ) {
     _instance = new ConcreteDevice;
  }
  return _instance;
}
```

Basically, the unique device is accessed exclusively through the *instance* member function. For example, the *deliver* method is invoked as:

```
ConcreteDevice::instance()->deliver();
```

There is no need to instantiate the device first (actually it cannot be), and no need to specify a particular name for the instance.

2.6 Analysis Summary

Starting from the preliminary MPI-1 Object Model, we identified and solved several key design issues using the design patterns. The expanded MPI-1 Object Model is shown in Figure 6. This establishes the fundamental structure for the object-oriented design of MPI-1, and is a main contribution of this work.

However, this design only represents one solution; other possibilities also exist. For example, in our design, we use aggregation and treat the class `Cache` as part of the class `Communicator`. But we can also think of "caching" facilities as additional responsibilities attached to a `Communicator`, so a more flexible way would be to follow the Decorator design pattern and to add the "caching" functionality dynamically as necessary. This choice could be debated in the detailed design stage. Our aggregation approach is simple and follows the MPI specification directly.

Overall, this is an architectural design; to carry out the implementation of MPI based on this design, there are many details that need to be added. For example, the `Transports` class represents the message-passing mechanism, but its exact interface needs to be specified, which could be a demanding task. `Transports` also has to provide some sort of resource management and to ensure progress [14]. Similarly, there are also many issues that need to be solved for the class `Message`. For instance, `Message` should contain enough information so that `Transports` can handle messages properly. This also means that the interfaces of `Message` and `Transports` should be carefully defined with considerations for each other. Furthermore, messsages have headers and object-oriented transports would be defined, if desired.

3 MPI++

The first attempt at an C++ interface for MPI occurred in 1994 with the C++ implementation, defined by some of us, called MPI++ [28]. For the implementation of MPI++, a medium weight object-oriented approach was taken. Although MPI++ is a class library, it only has a few classes not included in the MPI C++ binding. The requirements that guided the design and implementation of MPI++ are as follows:

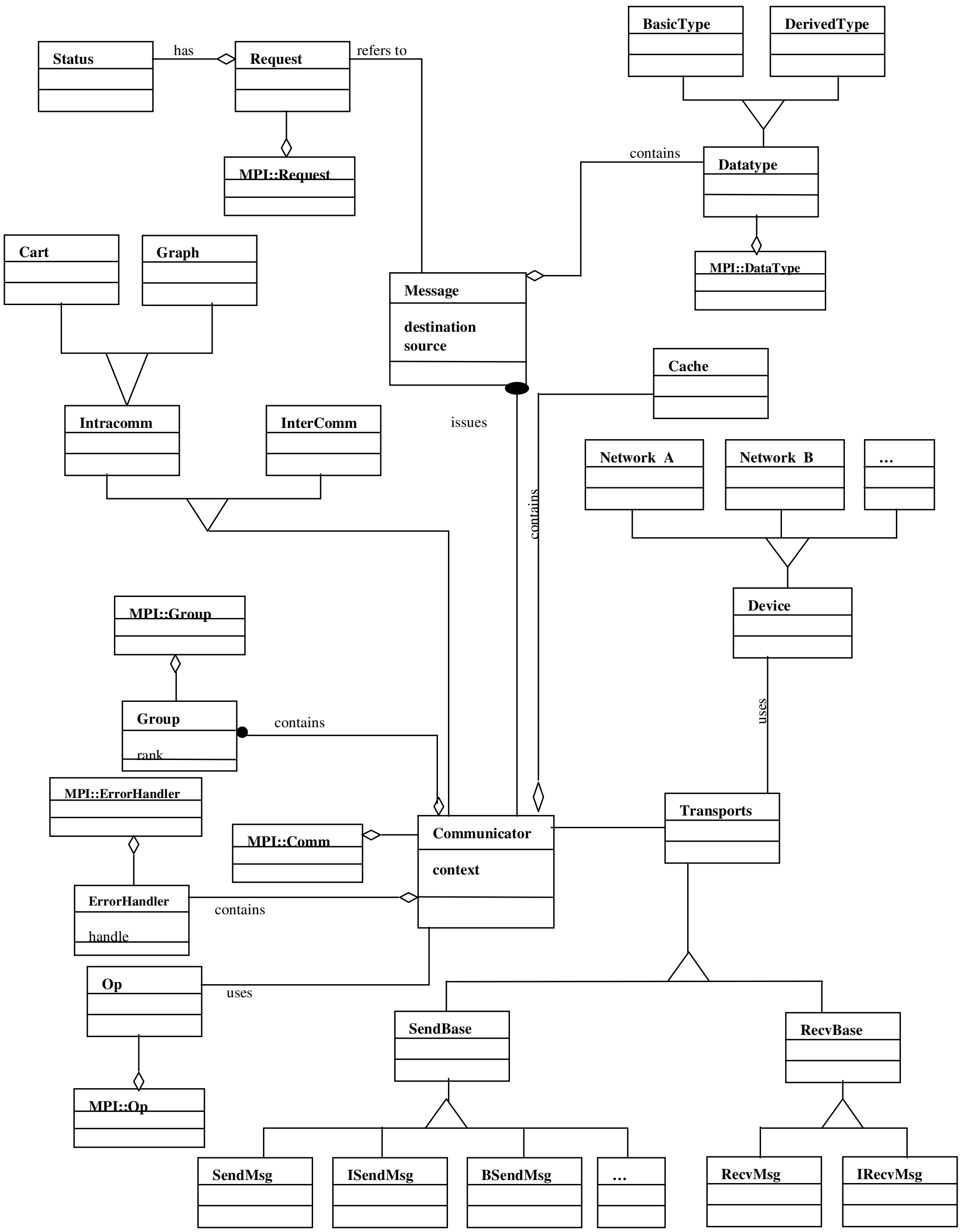

Figure 6: An Expanded MPI-1 Object Model

1. The MPI++ interface should be consistent with the MPI C interface.
2. Performance-portability should be comparable to that of the C interface. That is, MPI++ features should be lightweight whenever the corresponding C functions are lightweight.
3. MPI++ should exhibit a clean design that is prototypical of well-written object-oriented programs.

The following sections consider these requirements in turn, discussing the rationale behind them, and their influence on the design decisions.

3.1 The MPI++ Interface

Because of the standardization process, MPI's language-independent semantics are not subject to rapid change. It was thus attractive (as well as prudent) to follow the MPI standard carefully when designing and implementing MPI++, even though this is not an official language binding. MPI has a C language binding, of which many potential users of MPI++ already have a working knowledge. Therefore, little effort should be needed for such users to migrate from the C interface to MPI++. These considerations strongly suggested that the MPI++ interface be both syntactically and semantically consistent with the C interface. Semantic consistency requires that the counterparts in the two interfaces must perform the same function without unexpected side effects, including effects on performance. This prevents erroneous uses or unexpected performance penalties when one switches from one interface to the other.

When using the MPI++ interface instead of the standard C interface, many small differences must be noted. The first difference to note is that the statement `#include <mpi.h>` from the standard C interface has been replaced by `#include <mpi++.h>` in the MPI++ interface. Another change to notice is that the prefix `MPI_` has been changed to `MPIX_`. A prefix other than `MPI_` was needed in order to avoid name conflicts, since MPI++ was implemented on top of the C interface. [3] It is important to note that functions that do not readily fall into classes are not given wrappers and must be called using the standard C calls. These pragmatic steps were consistent with our earlier experimental pragmatic goals.

One big difference between the C interface and the MPI++ interface is that all MPI++ versions of the MPI functions are methods of an MPI++ class. For example, the function call in the C version:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

appears in the MPI++ version as a method of the object `MPIX_COMM_WORLD`:

```
MPIX_COMM_WORLD.Rank(&rank);
```

For example, the send and receive calls that, in the C code, take the communicator `comm1` as an argument:

```
MPI_Send(message, strlen(message), MPI_CHAR, size-1,TAG, comm1);
MPI_Recv(message, MAX_MESSAGE_SIZE,  MPI_CHAR, 0,TAG, comm1, &status);
```

become methods of `comm1`:

```
comm1.Send(message, strlen(message),MPIX_CHAR, size-1, TAG);
comm1.Recv(message, MAX_MESSAGE_SIZE, MPIX_CHAR, 0,TAG, &status);
```

One way to describe the relationship between the C binding and MPI++ is to say that we have moved the communicator argument to the front of the function call, making it the object to be dereferenced. This has let us drop the prefix "`MPI_`" or "`MPI_Comm`" from member function names. The class hierarchy described in the next section shows the value and effectiveness of this approach.

[3] Through the use of the C preprocessor, we provide shortcuts that allow the use of the `MPI_` prefix. However, we prefer to use the `MPIX_` prefix because MPI++ is not a standard interface of MPI, and only standard bindings may use the `MPI_` prefix.

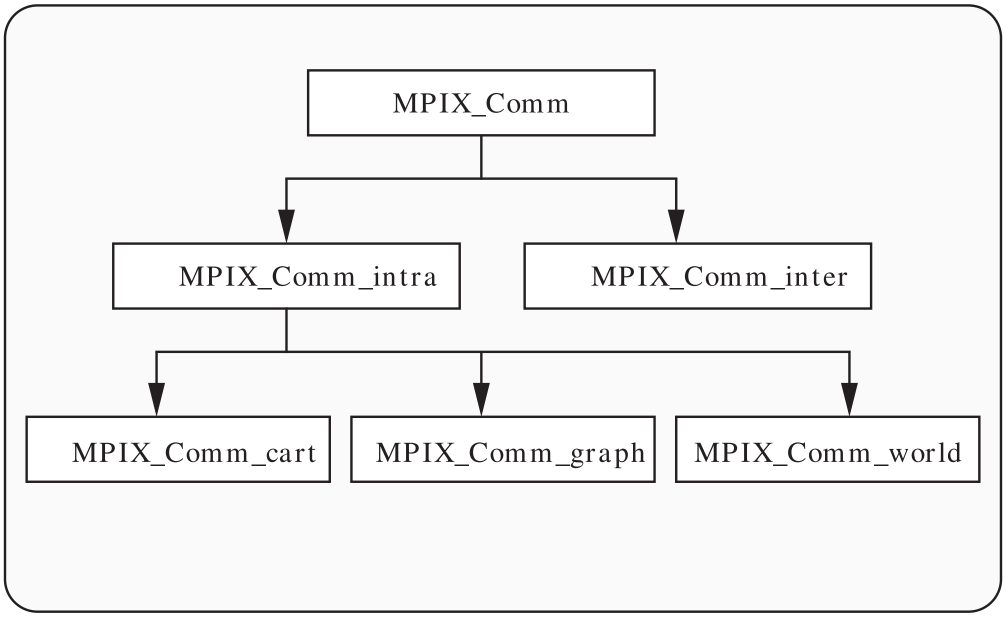

Figure 7: Hierarchy of Communicator-Based Classes for MPI++

3.2 Object-Oriented Design of MPI++

As discussed in Section 2, the MPI specification is object-based. It is therefore relatively easy to identify the major classes for MPI++. In general, class hierarchies can be either single-rooted or forests [31]. Either approach could have been used in the design of MPI++. The forest approach, in which a number of superclasses support the user-accessible objects, fits the MPI specification naturally. We were convinced that using it would result in a system that was more likely to be consistent with the C interface, and also initially easier to build. Most MPI functions are available as methods of a class in MPI++. However, some MPI functions such as `MPI_DIMS_CREATE` don't fall easily under one object. In MPI++, these functions are called using the standard C bindings (These functions were not given wrappers).

The Communicator-based Classes The communicator-based classes constitute the backbone of the MPI++ architecture. Their hierarchy is illustrated in Figure 7. The base class `MPIX_Comm` contains all the functions common to both intra- and inter-communicators, such as point-to-point communication and accessor functions. Its interface is sketched in Figure 8.

The `MPIX_Comm_intra` (intra-communicators) class, derived from `MPIX_Comm`, contains collective communication and topology functions, as well as various functions that do not apply to inter-communicators[4] A sketch of its interface is given in Figure 9.

The `MPIX_Comm_inter` (inter-communicators) class is also derived from the `MPIX_Comm` class. It contains only those functions peculiar to itself. Figure 10 gives a sketch of its interface.

While most of these methods are straightforward to use, users must note several points. First, the destructor for the `MPIX_Comm` class does not free the communication resources associated with the communicator. All instances of `MPIX_Comm` must therefore be freed explicitly, as in C, using `MPIX_Comm::Free()`[5]

Second, users must distinguish between using the overloaded "=" operator for communicators and instantiation via the communicator's copy constructor. For example, during copy construction, a call made by

[4]MPI-2 removes intercommunicator collective restrictions.

[5]The semantics of the C binding have been retained. This notionally follows the restrictive approach we applied when defining `operator=` as a shallow copy: we have chosen not to introduce implicit synchronization across process groups.

```
class MPIX_Comm {
 public:
  MPIX_Comm();                          // Constructor
  virtual \~ MPIX_Comm();            // Destructor
  virtual int Free();                   // Free
  virtual int Dup(MPIX_Comm*);          // Initializer
  // Environment
  virtual int Abort(int);
  virtual int Errhandler_set(MPI_Errhandler&);
  virtual int Errhandler_get(MPI_Errhandler*);
  // Accessors
  virtual int Size(int *);
  virtual int Rank(int *);
  // Sends
  virtual int Send(void *,int,MPI_Datatype,int,int tag=0);
  // Receives
  virtual int Recv(void*,int,MPI_Datatype,int,int,MPIX_Status*);
  // Pack and Unpack operations
  virtual int Pack(void*, int, MPI_Datatype, void *, int, int*);
  virtual int Unpack(void*, int, int*, void*,int, MPI_Datatype);
  // Overloaded operators (= does copying, not parallel duplication)
  virtual MPIX_Comm& operator=(const MPIX_Comm& old_comm);
  virtual int operator==(const MPIX_Comm&);
  virtual int operator!=(const MPIX_Comm&);
 private:
}
extern MPIX_Comm         MPIX_COMM_NULL;
extern MPIX_Comm_world   MPIX_COMM_WORLD;
```

Figure 8: Interface of the MPIX_Comm Class.

```
class MPIX_Comm_intra : public MPIX_Comm {
 public:
  // Constructors
  MPIX_Comm_intra(void);
  MPIX_Comm_intra(const MPIX_Comm_intra&);
  MPIX_Comm_intra(const MPIX_Comm_intra&, const MPIX_Group&);
  MPIX_Comm_intra(MPIX_Comm_intra&, int, int);
  // Initializers
  virtual int Create(const MPIX_Group&, MPIX_Comm_intra*);
  virtual int Split(int, int, MPIX_Comm_intra *);
  // Collective operations
  virtual int Barrier(void);
  // Topology functions
  virtual int Cart_create(int, int dims[], int periods[], int,
                          MPIX_Comm_intra*);
};
```

Figure 9: Interface of the MPIX_Comm_intra Class.

```
class MPIX_Comm_inter : public MPIX_Comm {
public:
  // Constructors
  MPIX_Comm_inter(void);
  MPIX_Comm_inter(const MPIX_Comm_inter&);
  // Accessors
  virtual int Remote_size(int*);
  virtual int Remote_group(MPIX_Group*);
  // Intracommunicator constructor
  virtual int Merge(int, MPIX_Comm_intra*);
};
```

Figure 10: Interface of the MPIX_Comm_inter Class.

all processes of the group is illustrated. This call causes a deep copy of the communicator to occur, which involves synchronization across the processes that made the call:

```
// communicator instantiation calls
// parallel call, deep copy occurs
MPIX_Comm_intra::MPIX_Comm_intra(const MPIX_Comm&) MPIX_Comm_intra c1=comm1;
```

By way of contrast, `operator=` is a local call: when pre-existing objects are assigned to, no parallel operations are done. We note that this is a shallow copy of the communicator.

```
// overloaded MPI_Comm::operator=(const MPI_Comm&) which
// does not create a duplicate of comm1
MPIX_Comm_intra c2; // initialized to be MPI_COMM_NULL
c2 = comm1;         // local call
```

In short, setting one `MPIX_Comm_intra` instance equal to another does not create a new communicator. Thus, only one of the two communicator instances above should call `MPIX_Comm_intra::Free()`. The user must decide how to manage this freeing operation themselves. This follows the logic that C operators, extended to C++ objects, should not have unexpected side effects. In practice, the above property can be avoided by using reference counting [8], but the real issue is the potential for deadlock. If `operator=` required loosely synchronous invocation over the entire group involved in the communication, subtle bugs involving synchronization could arise from seemingly local operations.

Figure 7 shows that three subclasses are derived from `MPIX_Comm_intra`: `MPIX_Comm_world`, `MPIX_Comm_cart`, and `MPIX_Comm_graph`. `MPIX_Comm_world` is special, in that there is exactly one instance of it in any MPI program. This instance is called `MPI_COMM_WORLD`. To prevent users from creating additional instances of this class, `MPIX_Comm_world` is designed and implemented as a Singleton [17]. This class adds two methods to `MPIX_Comm_intra`, which are used to manipulate the MPI environment:

```
int Init(int *argc, char*** argv)
int Finalize()
```

The other two subclasses derived from `MPIX_Comm_intra` provide support for virtual topologies. A virtual topology is a machine-independent naming abstraction used to describe communication operations in terms that are natural to an application. `MPIX_Comm_cart` supports cartesian topologies, i.e. topologies based on rectangular coordinates. `MPIX_Comm_graph` supports general graph topologies, in which processes are nodes of an arbitrary graph.

```
class MPIX_Group {
 public:
  // Constructor
  MPIX_Group(void);                        // create an empty group
  MPIX_Group(const MPIX_Group& old_group); // copy a group
  virtual ~ MPIX_Group(void);               // Destructor
  virtual int Free(void);                  // Free a group
  virtual MPIX_Group& operator= (const MPIX_Group&); // Assignment
  // Group accessors
  virtual int Size(int *);
  // Group manipulation
  virtual int Union(const MPIX_Group&, MPIX_Group*);
  // Overloaded set operators
  virtual MPIX_Group operator- (const MPIX_Group&);
  virtual MPIX_Group operator+ (const MPIX_Group&);
  // Overloaded self-modifying operators
  virtual MPIX_Group& operator-= (const MPIX_Group&);
  virtual MPIX_Group& operator+= (const MPIX_Group&);
  // Shift members off the ends of a group
  virtual MPIX_Group         operator<< (const int);
  virtual inline MPIX_Group  operator>> (const int);
  // Compare the members of groups
  virtual int operator== (const MPIX_Group&);
  virtual int operator!= (const MPIX_Group&);
 private:
  MPI_Group group;
};
extern MPIX_Group MPIX_GROUP_EMPTY;
extern MPIX_Group MPIX_GROUP_NULL;
```

Figure 11: Interface of the MPIX_Group Class.

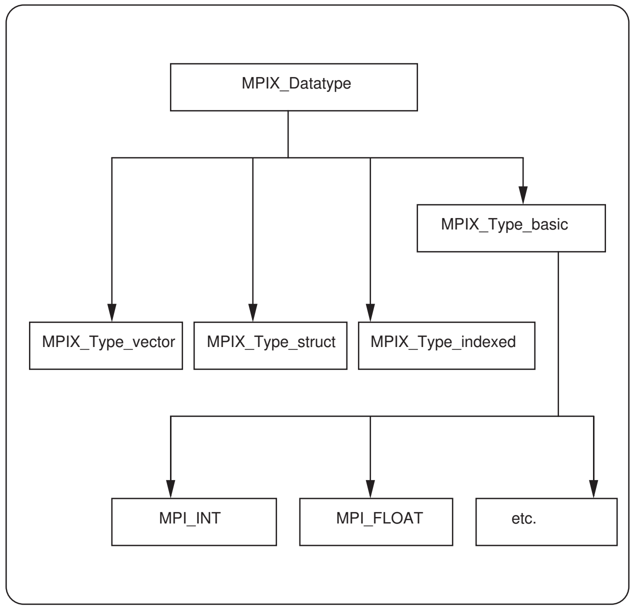

Figure 12: Hierarchy of Datatype-Based Classes in MPI++

The Group Class Figure 11 sketches the interface of the `MPIX_Group` class, most of which is straightforward. `MPIX_Group` objects are not explicitly used in communication; their only use in MPI-1 is for describing group members when building new communicators using `MPIX_Comm::Create()`. They are therefore not as important as the communicator classes in most MPI++ programs. However, they are more important in MPI-2, which has a dynamic process model using groups as well as communicators.

The Datatype Class In addition to the basic datatypes such as `int`, `float`, and `char`, MPI provides functions for building derived data types, including vector, indexed, and structured types. New data types are built recursively from previously defined datatypes. Figure 12 shows the hierarchy of datatype classes for MPI++.

As can be seen, the datatypes used in the MPI C binding fit into the object-oriented programming model without further abstraction. In our earlier MPI++ implementation, we had chosen not to make C++ wrappers for them.

The Handler Classes The four remaining classes in the MPI++ class hierarchy are used during non-blocking communications. They are `MPIX_Request`, `MPIX_Status`, `MPIX_Request_collection`, and `MPIX_Status_collection`. Their methods correspond to the appropriate MPI C functions that act on them. For example, the C function:

```
MPI_Wait(MPI_Request *request, MPI_Status *status);
```

becomes:

```
MPIX_Request request;
request.Wait(MPIX_Status *status)
```

Figure 13 shows a sketch of the interface for `MPIX_Request`; the interface for `MPIX_Status` is illustrated in Figure 14. The interfaces for `MPIX_Request_collection` and `MPIX_Status_collection` are similar. The methods for `MPIX_Request_collection` correspond to MPI C functions that apply to an array of type `MPI_Request`.

```
class MPIX_Request {
 public:
  // Constructor
  MPIX_Request();
  // Destructor
  virtual ~ MPIX_Request();
  // Request destructor
  virtual int Free();
  // Request operations
  virtual int Wait(MPIX_Status *);
  virtual int Test(int*, MPIX_Status*);
  // Overloaded operations
  MPIX_Request& operator=(const MPIX_Request&)
 private:
  MPI_Request request;
};
```

Figure 13: Interface of the MPIX_Request Class.

```
class MPIX_Status {
 public:
  MPIX_Status();                        // Constructor
  virtual ~ MPIX_Status();          // Destructor
  // Status accessors
  virtual int MPI_TAG();
  virtual int MPI_SOURCE();
  virtual int Test_cancelled(int *);
 private:
  MPI_Status status;
};
```

Figure 14: Interface of the MPIX_Status Class.

3.3 Comments on Design

From the above discussion, one can see that MPI++ offers most of the major functionality of MPI's C interface. The features currently missing, such as profiling and environmental inquiry functions, are mostly of an auxiliary nature, and could be added easily.

MPI++ employs several standard object-oriented concepts. First, MPI++ uses inheritance, or white box reuse. In particular, the communicator-based class hierarchy gives flexibility to, and promotes reusability and extensibility in, the design of parallel libraries. For example, if a particular application requires a two-dimensional Cartesian topology of dimension $P \times Q$, then the user can create one with minimal effort using inheritance and customize. A sample interface might look like:

```
class MPIX_Grid2d : public MPIX_Comm_intra {
public:
  MPIX_Comm_intra Row, Column;         // Row and column communicators
  MPIX_Grid2d(void);                   // Constructor
  ~ MPIX_Grid2d(void);                  // Destructor
  int Free(void);                      // Free grid
  int Dup(MPIX_Grid& grid_out);        // Duplicate grid
  // Initialize grid
  int Init(MPIX_Comm_intra& comm_in, int P, int Q);
  // Overloaded point-to-point operations
  int Send(void*, int, MPIX_Datatype, int P, int Q, int tag);
  int Recv(void*, int, MPIX_Datatype, int P, int Q, int tag,
           MPIX_Status&);
  // Overloaded collective operations
  int Bcast(void*, int, MPIX_Datatype, int P, int Q);
  int Reduce(void*, void*, int, MPIX_Datatype, MPI_Op, int P, int Q);
  // Grid accessors
  int P(void);
  int Q(void);
  int p(void);
  int q(void);
 private:
  int P_, Q_; // dimensions of grid
  int p_, q_; // local processes' position in grid
};
```

The row communicator would contain all the processes in the same logical row of the grid, and would be used for communication local to the row. The column communicator would contain the processes sharing in the same logical column, and would be used analogously. Point-to-point functionality is overloaded to accept grid-specific process names, such as coordinate `(5,3)`, instead of process ranks. This is because our design takes advantage of the underlying object-based design of MPI. The resulting MPI++ design, looked at as a class hierarchy, is thus relatively shallow and wide in terms of the C++ objects, and little overhead is consequently introduced.

Polymorphism is another key feature of object-oriented design. Methods in MPI++ classes are declared `virtual` in conformance with the Inheritance Canonical Form, in order to support reuse by users of the classes. Thus, MPI++ was designed for simple inheritance by users, though the implementation of MPI++ itself does not currently support virtual inheritance. We expect, in future, to exploit inheritance within the implementation to support a better design for MPI datatypes.

Finally, MPI++ uses operator overloading and default arguments to provide simpler and more intuitive member functions. For example, MPI++ overloads the `-` operator to have the same effect as the `MPIX_Group::Difference` member function. Other examples include the `|` and `&` operators for `MPI_Group::Union` and `MPIX_Group::Intersection` respectively, and the use of `<<` and `>>` to shift a specified number of group members off the front or end of a group.

Furthermore, the tag argument in communication calls is usually an optional argument, which MPI++ sets to 0 by default. This simplifies programs that do not use tags.

4 OOMPI

OOMPI was introduced after MPI++, and offers an alternative, higher abstraction approach to accessing MPI from C++. For instance, a typical MPI function call in C is of the following form:

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

Here, i, 1, MPI_INT, and tag specify the content and type of the message to be sent, and comm and dest specify the destination of the message. A more natural syntax results from encapsulating the pieces of information that make up the message and the destination. That is, we could perhaps encapsulate i, 1, MPI_INT, and tag as a message object[6] and comm and dest as a destination (or source) object.

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

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

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

4.1 Ports and Communicators

Using an OOMPI_Port, we can send and receive objects with statements such as:

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

The OOMPI_Port object contains information about its MPI communicator and the rank of the process to whom the message is to be sent. Note, however, that although the expression Port.Send(i) is a very clear statement of what we want to do, there is no explicit construction of a message object. Rather, the message object is implicitly constructed (see 4.2 below).

Port objects are 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. This is similar to the design concepts in MPI++.

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

[6] The newer messaging standard, MPI/RT specifies this consistently, both by merging memory descriptions(buffer objects), and source/destination/communication(channels).

```
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.

4.2 Messages

We define an OOMPI_Message object with a set of constructors, one for each of the MPI base datatypes. 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 datatypes are declared, an OOMPI_Message object will be constructed automatically (and transparently) whenever a communication function is called with one of the base datatypes.

Message objects could be eliminated entirely by declaring each communication function in terms of every base datatype. 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 as follows:

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

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

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

Again, no explicit message is constructed.

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

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

The declaration of OOMPI_Port::Send() is

```
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 OOMPI_NO_TAG is not the tag used on the message. Rather, it is a dummy value that indicates that no tag was explicitly given, so inside the body of OOMPI_Send(), a default tag is used, depending on the type of the data. OOMPI reserves the top OOMPI_RESERVED_TAGS tags. Users can use any tag between zero and OOMPI_TAG_UB (This is similar to choices offered by MPI++).

4.3 User Defined Datatypes

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

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

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

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

```
class foo : public OOMPI_User_type {
public:
  foo() : OOMPI_User_type(type, this, FOO_TAG) {
    // Build the datatype 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 as follows:

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[7].

A necessary condition 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, MPI_Type_Struct() and MPI_Type_commit() are called to build the datatype, and the up is performed on the mutex.

4.4 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.

4.5 A Streams 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, for convenience, even though MPI is message and not stream based itself That is, messages can be sent and received in OOMPI with the statements:

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

Since second arguments to operator>>() and operator<<() are disallowed, default tags (based upon the message type) are used. These default tags can be overridden, however. (See section 4.10 for the discussion of the OOMPI_Message and OOMPI_Array_message objects.) Note that user-defined datatypes 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(). This adds overhead in return for simplicity.

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 (unless OOMPI_Ports are localized to a single thread) since they imply that the OOMPI_Port object must contain local state. OOMPI could demand that threads must keep their own copies of ports, a break in the similarity between ports and send.

4.6 Packed Data

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

[7] The functions Get_tag() and Set_tag() are inherited from the OOMPI_Tag class.

```
int i;
OOMPI_Port Port;
OOMPI_Packed msg(OOMPI_COMM_WORLD.Pack_size(i, 400),
                 OOMPI_COMM_WORLD, PACK_TAG);
...
msg.Start();
for (i = 0; i < 200; i++)
  msg << i << rank;
msg.End();
Port << msg;
```

The arguments to the `OOMPI_Packed` constructor are the size of the buffer to be created, the communicator, and the tag to be used for sending and receiving this instance. Note that no `count` argument is passed to the `Port` when sending the object; an `OOMPI_Packed` object inherently knows its count. That is, sending an `OOMPI_Packed` object will send as many bytes as were packed. Receiving an `OOMPI_Packed` object will attempt to receive a message as long as the entire buffer. MPI-1 allows the normal receipt of a shorter-than-expected message.

Note that the `OOMPI_Packed` object has local state. However, it does not appear to be a common case for more than one thread to pack into the same buffer. Therefore, we define a process that has multiple threads packing into one buffer to be erroneous, and avoid locking concepts and locked objects. 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.

4.7 Attributes

Unlike MPI++, 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 (Since this is a form of object declaration, users can exploit inheritance).

4.8 Objects

Listed below are all the objects that are used by OOMPI at the user level. Each object contains a brief description and list of functional requirements.

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

4.9 Communicator Objects

The objects associated with communicators are as follows:

`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 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.

The OOMPI_Group object encapsulates all the operations on groups. In OOMPI, the OOMPI_Group is used by the OOMPI_Communicator object.

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

4.10 Message and Data Objects

The OOMPI objects associated with messages are as follows:

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 datatypes (base C++ types and user-defined types) into one object type. That is, every MPI-1 function that includes a (void *) data buffer argument is replaced with an OOMPI_Message argument (and/or OOMPI_Array_message argument, see below). Since the OOMPI_Message object includes the MPI datatype and a pointer to the top of the data, functions that have OOMPI_Message arguments inherently know the data's type and where it resides in memory.

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

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

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

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

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

That is, there are three mechanisms to pass data of base C++ types (user defined types are discussed in section 4.3) to OOMPI functions: two implicit mechanisms and one explicit mechanism. OOMPI_Message and OOMPI_Array_message are used to implicitly promote the base types. Note that the implicit promotion to an OOMPI_Array_message is not sufficient for the stream interface because the count argument cannot be supplied.

```
int i, j[10];
OOMPI_Port Port;
...
// Both implicit mechanisms can be used with the
// standard interface:
Port.Send(i);
Port.Send(j, 10);
// Only the implicit scalar promotion can be used with
// the stream interface:
Port << i;
```

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

```
int i, j[10];
OOMPI_Port Port;
OOMPI_Message imsg(i, MY_INT_TAG);
OOMPI_Message jmsg(j, 10, MY_INT_ARRAY_TAG);
...
// Explicitly formed messages can be sent through the
// standard interface:
Port.Send(imsg);
// Or they can be sent through the stream interface:
Port << jmsg;
// They can also be re-used:
i++;
j[3]++;
Port << imsg << jmsg;
```

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

The OOMPI_User_type object is the heart of user-defined datatypes. It must be inherited and initialized by all objects that will be sent and/or received in message passing calls. 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.

4.11 Object Semantics

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

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

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

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

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

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

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

```
int i = 0;
OOMPI_Request 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 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:

```
OOMPI_Cart_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 (e.g., `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.

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.

`const` Semantics. The const version of OOMPI was specifically designed to be implemented on top of existing MPI-1 ANSI C bindings. As such, it was onerous 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 includes C++ bindings which make use of `const`, future versions of OOMPI will 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 the current OOMPI implementation 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.

5 MPI C++ Language Bindings

A number of proposals for the MPI C++ bindings were introduced during the course of the MPI-2 Forum. The original (preliminary) proposal was modeled closely after the MPI++ class library (Section 3). The initial proposal introduced a major question to the Forum; specifically whether the bindings should be a full-blown class library or should they be something closer to the C interface? Both options were explored, with proposals for each introduced over a period of time. After the Forum had a chance to study and evaluate the class library proposal, it was decided that the role of the C++ bindings was to facilitate the development of class libraries, but not actually to be a class library. The proposed class library later became Object-Oriented MPI (OOMPI), see Section 4).

After the class library approach was discarded, the pendulum swung towards conservatism and a proposal for low-level bindings was made. These proposed bindings were close to the C bindings, but provided a few C++ features such as `const` and reference semantics. However, the Forum decided that these bindings were too low-level and did not do enough to enable class library design.

Thus, the final, and accepted, proposal for MPI C++ bindings found the middle ground between big and small. The bindings contain a number of class library-like features, but still remain limited enough not to constrain class libraries built using them. In many respects, it very closely resembles the original MPI++ design (Section 3).

5.1 Object-Based Design

An obvious choice for the C++ bindings (once there was a decision to go with the "small" interface) was to turn the handles into regular C++ objects. These objects, however, retained the same handle-based semantics as their C counterparts. Namely, the C++ objects are user-level handles to underlying implementation-dependent objects. Thus, the C++ bindings present the visible portion of the Bridge pattern discussed in Section 2.

Most MPI functions became methods associated with MPI objects. In most cases, which object to associate with a given function was "obvious", though the rationale is ultimately more intuitive than rigorous. Examples of these obvious choices were `Comm::Send()` and `Request::Wait()`. Some functions were "obvious" candidates for a specific class even though they did not include a single IN or OUT argument of that type. For example, `MPI::Datatype::Create_struct()` takes an IN array of `MPI::Datatype`. Such functions were still defined on that class, but were declared `static` since they have no corresponding `this` pointer.

5.2 Naming

MPI-1 did not use consistent naming rules. Often, names are of the form `MPI_Object_action` as in `MPI_COMM_SPLIT` and `MPI_INTERCOMM_MERGE`, but sometimes they are not, e.g., `MPI_TYPE_CONTIGUOUS`. Sometimes the verbs are consistent, e.g., `MPI_COMM_FREE` and `MPI_TYPE_FREE`, but sometimes they are not, as in `MPI_ERRHANDLER_SET` and `MPI_ATTR_PUT`.

Unlike MPI-1, MPI-2 uses a consistent naming scheme of the form `MPI_Object_action_subset`. For the C++ bindings, the Forum decided to use the consistent names. Although the MPI-2 C++ scheme for both MPI-1 and MPI-2 functions are slightly different from the MPI-1 C names in several cases, the consistent C++ naming scheme was found to be advantageous for the following reasons:

1. The C++ bindings are new. There is no existing code that needs to be changed or documentation that must be rewritten.
2. It was agreed that using the inconsistent names was more of a problem in a C++ context where the structure highlights discordant design. The inconsistent names would result in C++ names such as `Status.Get_count()` and `Status.Get_elements()`, which both use the verb "get," while `Comm.Size()` and `Comm.Rank()` do not.[8]
3. The C++ names are necessarily different from the C names already, e.g., `Comm.Send()` instead of `Comm.MPI_Send()`.

One relatively new feature of ANSI C++ is the `namespace` construct which allows programs to provide explicit scoping of MPI names. The MPI C++ bindings make use of this feature by including all names within the scope of a `namespace` MPI.[9] As such, all C++ MPI names are prefixed with "`MPI::`".

5.3 Object Semantics

Constructors and Destructors. MPI-1 has routines that clearly create objects (e.g., `MPI_COMM_DUP`) and routines that free them (e.g., `MPI_COMM_FREE`). It seems at first natural in C++ to turn these and related functions into constructors and destructors. The main problem with such an approach is that `Create()` and `Free()` are collective operations. Thus a declaration

```
MPI::Comm a(MPI::COMM_WORLD)
```

intended implicitly to `MPI_COMM_DUP` the predefined `MPI_COMM_WORLD` communicator would be a "collective declaration." Worse, the return from the routine where this variable was declared would be a collective operation, when the object was implicitly freed with `MPI_COMM_FREE` in the destructor.

MPI-2 therefore chooses a path that may be unfamiliar to C++ programmers: the application is responsible for explicitly creating and freeing objects using the appropriate explicit MPI function calls. Consistent with the MPI memory management model, memory management is not automatically handled by constructors and destructors.

Nevertheless, MPI-2 specifies default constructors that initialize objects to be equivalent to their corresponding `MPI::*_NULL` handles, and destructors that do free the "top-level" C++ object, but not the underlying object to which it refers.

[8] In comparison, the actual names of the last two functions use the standardized verb "get". They are `MPI::Comm::Get_size()` and `MPI::Comm::Get_rank()`, respectively.

[9] Some C++ compilers do not implement the `namespace` construct yet. An *Advice to implementors* in the MPI-2 standard allows implementors to use a non-instantiable `MPI` class if `namespace` is not available.

Copy and Assignment. Since the C++ objects are still handles to underlying objects, the copy and assignment operations are shallow. The assignment

```
MPI::Comm comm = MPI::COMM_WORLD;
```

does not create a new communicator, `comm` is now an alias for `MPI::COMM_WORLD`. That is, `comm` and `MPI::COMM_WORLD` now reference the same underlying object. The `MPI::Status` object does not follow this rule. Since `MPI::Status` has public data members and does not necessarily point to internal implementation-dependant data, copies and assignments are deep.

Comparison. Similarly, comparisons of MPI handles return true only if they point to the same internal object. `MPI::Status` is again an exception to this rule; the comparison operators are not defined on the `MPI::Status` class because it is not a handle to an underlying object.

Constants. The predefined MPI constants are singleton objects, meaning that they can only be instantiated once. In contrast to the C and Fortran constants, their types are explicitly specified with the exception of `MPI::COMM_NULL` (discussed below). All constants must also be `const` to allow for possible compiler optimizations, particularly when passed as function parameters.

5.4 The `Comm` Class Hierarchy

The C++ bindings recognize the inheritance relationships between types of communictors by defining the following four communicator classes: `Intercomm`, `Intracomm`, `Cartcomm`, and `Graphcomm`. `Intercomm` and `Intracomm` are derived from the `Comm` base class; `Cartcomm` and `Graphcomm` are derived from `Intracomm`. Functions that require specific types of communicators (e.g., `MPI_CART_RANK`) are defined on their respective classes, while functions that apply to all types of communicators (e.g., `MPI_SEND`) are defined on the base class, `MPI::Comm`.

The function `MPI_COMM_DUP` presents a unique problem in this `MPI_COMM_DUP` returns a new communicator of the same type as the original. In C, this is not a problem because all four communicators are of the same type. In C++, however, each communicator is a different type, and one function cannot return four different types. Also, since it is not desirable to have copy constructors that perform collective actions, `DUP` must be realized as a regular member function. Several alternatives were proposed:

1. *Return by Reference.* `virtual MPI::Comm& MPI::Comm::Dup()` This binding does not fit the MPI memory management model that requires the user to manage memory. Thus, this C++ version of `MPI_COMM_DUP` would be significantly different than its C and Fortran counterparts if it were to allocate memory itself.
2. *Return by INOUT.* `virtual void MPI::Comm::Dup(MPI::Comm& newcomm)` Although this binding would not break the MPI memory management model, the syntax is non-intuitive, and does not conform to ideas discussed later in Section 5.6
3. *Return by value.* `COMMTYPE MPI::COMMTYPE::Dup()` Substituting any of the four communicator types for `COMMTYPE` gives four non-`virtual` bindings; this function is not implemented on the `MPI::Comm` base class. However, typical parallel library functions take any type of communicator as an argument, duplicate it (regardless of its type), and use it to perform simple sends and receives. That is, a typical function in a C++ parallel library may be prototyped as:

   ```
   void startFoo(Comm& usercomm)
   ```

   But since `Dup()` is not defined on the `Comm` base class, `startFoo` cannot duplicate `usercomm` without first casting it to another type, which defeats the point of prototyping the argument as a `Comm&`. Simply put, this binding restricts the use of ploymorphism.

The Forum decided that option 3 was the best solution since it does not break the MPI memory manangement model, has intuitive syntax, and returns the correct type. To provide a “virtual `Dup()`”, a new function was introduced that only exists in the C++ bindings, `Clone()`:

```
virtual MPI::Comm& MPI::Comm::Clone() = 0;
```

Although `Clone()` does not conform to the MPI memory management model, the Forum decided that its lack of symmetry was acceptable because the name only exists in the C++ bindings. Note: that the prototypes of `Clone()` in the derived classes are slightly different; they return references to their respective (derived) types.[10]

The type of `MPI::COMM_NULL` is implementation dependent; it must be able to be used in comparisons and initializations with all types of communicators.

5.5 Exceptions

The C bindings for almost all MPI functions (except `MPI_WTICK` and `MPI_WTIME`) return an error code. In principle, an application can check this error code and take some action if there is an error. In practice, this error code is rarely used. First, by default, errors cause an MPI program to abort (this is often the desired behavior). Second, even if MPI is configured to return errors to the application, the MPI standard states that the state of a program is undefined after an MPI error. About the only thing an application can (semi)reliably do is print an error message and abort. Finally, it is tedious to check the return value from every MPI function call and to appropriately handle the errors.

C++ exception handling provides an elegant mechanism to handle errors. C++ applications are given the option of setting the default error handler to `MPI::ERRORS_THROW_EXCEPTIONS`, in which case MPI functions throw a C++ exception when there is an error. Thus, C++ member functions do not return error codes as function values.

5.6 Return Values

In C and in Fortran, values are returned through the argument list. For instance in C,

```
MPI_Comm newcomm;
MPI_Comm_dup(MPI_COMM_WORLD, &newcomm);
```

returns a new communicator in `newcomm`. Part of the reason for this is that the return value of the function is reserved for the error code. Since C++ MPI methods do not return error codes, the function return value is freed up to hold more than an error code, thus allowing more natural notation such as

```
MPI::Comm newcomm = MPI::COMM_WORLD.Dup();
```

In many MPI functions there is a single "OUT" quantity that makes sense as a return value in the C++ case. In other functions, the OUT quantity may not be readily returned (e.g., when it is an array – because of MPI's memory management model – or when there are multiple OUT arguments and it is not obvious which argument should be returned), or there may be no OUT quantity at all. In these cases, the corresponding C++ bindings return `void`.

5.7 References, Pointers, and `MPI_STATUS_IGNORE`

The MPI C++ bindings use `const` and reference semantics when possible. All "IN" parameters that are MPI objects are both `const` and passed by reference to allow for compiler optimization. Additionally, passing by reference does not incur the additional overhead of copy constructors. Thus, the binding for `MPI_COMM_SEND` is

```
void Comm::Send(..., const Datatype& datatype, ...)
```

[10] Not all C++ compilers implement `virtual` functions in derived classes that can overload the return types. An *Advice to Implementors* in MPI-2 allows implementors to return `MPI::Comm&` if their compiler does not yet support this feature.

The only pointer arguments are `char*` arguments for strings (because of convention) and `void*` arguments for choice buffer arguments.

This introduces a problem with the the new MPI-2 option to ignore a returned `MPI_Status` by specifying the constant `MPI_STATUS_IGNORE` for the corresponding OUT argument. In C, this constant is an argument of type `MPI_Status*`, and usually has the value `NULL`. In C++, the corresponding status argument is passed by reference and therefore must be a valid `MPI::Status` instance; it is not possible to pass a `NULL` pointer. Therefore, the C++ bindings take a different route, which is to have two bindings for every function with an OUT `MPI_Status` argument in the language-independent specification. One binding has a reference to a `MPI::Status` argument and the other has no `MPI::Status` argument. The C constant `MPI_STATUS_IGNORE` has no corresponding constant in C++.

5.8 Interfacing with C

To provide a transparent interface between C and C++, three functions are defined on all C++ MPI classes (except `MPI::Status`): a casting operator to cast C++ objects into C handles, a promotion operator to create C++ objects from C handles, and an assignment operator to allow the assignment of C handles to C++ objects.

There is no mechanism to translate directly from C++ to Fortran. A user must convert a C++ object into a C handle and then use the provided MPI-2 functions to convert it to a valid Fortran handle.

5.9 Design Details

An abbreviated definition of the `MPI` namespace and its member classes is as follows:

```
namespace \MPI/ {
  class Comm                          {...};
    class Intracomm : public Comm      {...};
    class Graphcomm : public Intracomm {...};
    class Cartcomm  : public Intracomm {...};
    class Intercomm : public Comm      {...};
  class Datatype                      {...};
  class Errhandler                    {...};
  class Exception                     {...};
  class File                          {...};
  class Group                         {...};
  class Info                          {...};
  class Op                            {...};
  class Request                       {...};
    class Prequest : public Request    {...};
    class Grequest : public Request    {...};
  class Status                        {...};
  class Win                           {...};
};
```

These include MPI-2 objects not previously considered in this paper and beyond the scope of the current discussion. Multiple and virtual inheritance are *not* used in the design. All member functions are `virtual` except those which are `static` (which cannot be `virtual`) and the `MPI_COMM_DUP` variants (which are implemented separately on each class).

6 Comparisons

Each of the three designs for MPI are valid object-oriented approaches, but are intended for different uses. Both OOMPI and MPI++ can be extended to include new MPI-2 features as well, they are not shown here. The "minimalistic" approach of the C++ bindings only provides an object-oriented base for MPI, and is intended to be used to build higher-level abstractions, (such as MPI++ and OOMPI).

Conversely, OOMPI makes use of many C++ features and does not attempt to preserve the function signatures of MPI. Since this backwards compatibility was not one of the goals, OOMPI had greater flexibility

	MPI++	OOMPI	C++ bindings
Use inheritance	Yes	Yes	Yes
Designed for inheritance	Yes	Yes	Yes
OUT arguments	Pointer	Reference	Reference
Return Values	Error Codes	OUT	OUT
Error reporting	Return code	Exceptions	Exceptions
Destructors	Empty	`MPI_*_free()`	Empty
Copy constructor	Deep	Shallow	Shallow

Table 2: Comparison between several points of the MPI C++ bindings, MPI++, and OOMPI.

in design which allowed for new abstractions, efficent usage of default arguments, and overloaded functions. If the MPI C++ bindings and OOMPI can be considered the two extremes of object-oriented design, MPI++ can be considered a hybrid of the two.

Table 2 shows a comparison of several points between the three approaches. The use of inheritance is a key difference between the class libraries and the bindings. For example, MPI++ and OOMPI both have expanded class hierarchies for communicators while the C++ bindings only have a single `MPI::Comm` class. However, all three libraries are designed *for* inheritance. That is, users can derive their own objects from objects in any of the three libraries.

The C++ bindings and OOMPI both use reference semantics for OUT variables, while MPI++ uses pointer semantics. The use of reference semantics allows for cleaner notation and for user-defined polymorphism as well as preventing overhead from copy constructors. Pointer semantics remain compatible with the MPI C bindings and potentially ease migration for C programmers, but provide for less elegant notation, which we regret in retrospect.

As a design requirement, the C++ bindings must conform to the C and Fortran 77 bindings and return integer result codes from function calls. MPI++, also by design, returns integer result codes. OOMPI, however, usually returns the OUT variable; exceptions are used to return errors. This allows for much more natural programming and readable code. For example:

```
  int rank1, rank2, rank3;

  MPI::COMM_WORLD.Rank(rank);     // C++ bindings
  MPI_COMM_WORLD.Rank(&rank);     // MPI++
  rank = OOMPI_COMM_WORLD.Rank(); // OOMPI
```

The OOMPI version is an assignment to the `rank` variable.

Since one of the MPI philosophies is to have the user manage memory space, the C++ bindings do not attempt to take advantage of destructors; destructors do not free the MPI handle in the underlying implementation. MPI++ also takes this approach. OOMPI uses intelligent reference counting to free the internal MPI handle when the last OOMPI object referencing it is destroyed. While taking the burden of memory management off the user, it can produce unexpected results if the programmer is not careful. The following OOMPI fragment shows this:

```
void foo()
{
  OOMPI_Intra_comm bar;
  bar.Dup(OOMPI_COMM_WOLRD);
}
```

When `bar` goes out of scope at the end of `foo`, its destructor is invoked, which triggers a call to MPI_COMM_FREE, which may cause collective communication across the `bar` communicator.

MPI++ takes the opposite approach in copy constructors; MPI_COMM_DUP is invoked, while the C++ bindings and OOMPI simply copy the handle. Unexpected collective communication can occur in the following MPI++ fragment:

```
void foo(MPI_Comm_intra bar)
{
}
```

When the `foo` is invoked, the calling communicator is MPI_COMM_DUP'ed into `bar`. Additionally, it is *not* MPI_COMM_FREE'ed when `bar` goes out of scope at the end of `foo`. This can result in wasted resources if a programmer is not careful.

6.1 Code Examples

Figure 15 shows an implementation of the canonical ring program implemented in the MPI C++ bindings. The ring program starts by having the highest numbered rank send a message to rank 0. When each rank receives the message, it passes the message to the next rank (i.e., the process with rank equal to `(my_rank + 1)`, or rank 0 for the highest numbered rank). The message passes around the ring this way `count` times, with rank 0 consuming the final message sent at the end of the program.

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

int main(int argc, char *argv[])
{
  MPI::Init(argc, argv);
  int msg[1] = { 123 };

  int rank = MPI::COMM_WORLD.Get_rank();
  int size = MPI::COMM_WORLD.Get_size();
  int to   = (rank + 1) % size;
  int from = (size + rank - 1) % size;

  if (rank == size - 1)
    MPI::COMM_WORLD.Send(msg, 1, MPI::INT, to, 4);
  for (int i = 0; i < 5; i++) {
    MPI::COMM_WORLD.Recv(msg, 1, MPI::INT, from, MPI::ANY_TAG);
    MPI::COMM_WORLD.Send(msg, 1, MPI::INT, to, 4);
  }
  if (rank == 0)
    MPI::COMM_WORLD.Recv(msg, 1, MPI::INT, from, MPI::ANY_TAG);

  MPI::Finalize();
  return 0;
}
```

Figure 15: Canonical ring program written with the MPI C++ bindings.

Note that MPI_INIT and MPI_FINALIZE are invoked as members of the non-instantiable `MPI` class, `MPI::Init()` and `MPI::Finalize()`, respectively. MPI_COMM_RANK, MPI_COMM_SIZE, MPI_SEND, and MPI_RECV, are implemented as member functions of MPI::COMM_WORLD — `Rank()`, `Size()`, `Send()`, and

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

main(int argc, char *argv[])
{
  MPI_Status status;
  int rank, size, msg[1] = { 123 }, count = 5, i = 0;

  MPI_COMM_WORLD.Init(&argc, &argv);
  MPI_COMM_WORLD.Rank(&rank);
  MPI_COMM_WORLD.Size(&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)
    MPI_COMM_WORLD.Send(msg, 1, MPI_INT, to);

  for (i = 0; i < count; i++) {
    MPI_COMM_WORLD.Recv(msg, 1, MPI_INT, from, &status);
    cout << "Node " << rank << " received " << msg[0] << endl;
    MPI_COMM_WORLD.Send(msg, 1, MPI_INT, to);
  }

  if (rank == 0) {
    MPI_COMM_WORLD.Recv(msg, 1, MPI_INT, from, &status);
    cout << "Node " << rank << " received " << msg[0] << endl;
  }

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

Figure 16: Ring program written with MPI++.

Recv(), respectively. The function signatures are similar to their C counterparts; the main difference being the missing communicator argument (it has become this).

The MPI++ implementation of the ring program is shown in Figure 16. It is very similar to the C++ bindings implementation except that MPI++ does not have a separate MPI:: namespace, and MPI++ uses default arguments for tags.

```
#include <iostream.h>
#include "oompi.h"

int
main(int argc, char *argv[])
{
  int count = 5, i = 0, msg = 123;

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

  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;
  }

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

Figure 17: Ring program written with OOMPI.

Figure 17 shows the OOMPI implementation of the ring program. While the structure of the OOMPI version is almost identical to the previous two programs, the invocation of the message passing calls is quite different. operator[] is used to specify the OOMPI_Port to send to or receive from. The Send() and Recv() calls are quite different as well; the integer message is transparently promoted into a OOMPI_Message and given a default tag before the communication takes place[11].

[11] Note that strategic use of inline in OOMPI drasticly reduces the amount of overhead that one would expect from the discussion of how the Send() and Recv() functions are invoked in this example.

7 Future Design Work

An object-oriented programming model would contribute to the understanding of the application programmer interface, offer intuition for the design of object-oriented support for such programming, and motivate strategies for extensions that provided added functionality and/or higher achievable performance. Key design isuues have been identified and solved providing an object-oriented architectural model for MPI-1. However, the added functionality and complexity of MPI-2 will require more rigorous analysis methods. These expanded methods will offer a more understandable, clearly defined and extensible model for the system. Conceptual consistency of a detailed model promotes better understanding of the system which will lead to better understanding of the implementation issues of the system. Extensibility offers better opportunities for studying changes in the system. Added services and/or possible avenues for optimization could be the source of the changes. The potential effects on the underlying object structures and interactions can then be more carefully studied by using a more rigorous approach.

Several opportunities for extending the analysis exist. The first extension is to give a more formalized view of the objects beyond the data dictionary. For instance, CRC cards would clearly identify the objects, responsibilities and collaborators [4]. State transition diagrams present a more in-depth analysis of the interaction of the objects within this system thus modeling more detailed behavior for the system [4]. Fowler [16] has presented a different approach, in that the domain is modeled conceptually. Using this approach to model the domain knowledge provides a smoother transition to the specification model for the system.

The collection of patterns is extensive and this repository of expertise may offer proven solutions. Buschmann [6] has organized the patterns into three levels, architectural, design, and idioms (implementation). For instance, an existing pattern could be chosen instead of created, as was done in this paper. The Broker pattern, which has a message-passing variant [6], or Streams, a variant of which is used in a push-driven message model [13] are both potential architectural patterns for the push-driven model of the MPI system [30]. In fact, a merger of patterns may need to be considered for the MPI-2 system. The Half-Sync/Half-Asynch architectural pattern may need to be merged with the Broker or Streams patterns to provide the communications and file I/O for MPI-2 [26]. Coordination of the architectural patterns with supporting design patterns is another benefit of reusing an architectural pattern [6, 13]. For instance, the Proxy is a complimentary design pattern used by the Broker architectural pattern to transparently provide appropriate message facility, such as IPC or network messages to applications [6] . Zimmer [32] and the Gang of Four [17] offer a different classification schemes of patterns as that provide more insight to combinations of patterns that provide design solutions. Zimmer's classification scheme concentrates on the relationships between patterns. One of the possible criteria of classification show which patterns are frequently used by other patterns in their solutions. For example, the command pattern may use the composite pattern. The Gang of Four takes yet another approach and classifies the patterns according to the problem that it will satisfy, such as behavioral or structural, etc [17].

Future study will look more closely at the MPI-2 domain and what is needed to provide a more detailed model of that system. Patterns and research on combinations of patterns provides an extensive store of techniques that may be reused to build this model for MPI-2. An implementation independent model can then be built and used to provide a means of understanding MPI-2. Applying the understanding gained through the use of these models, implementation issues and/or other issues can then be examined more readily for each specific environment.

8 Object-oriented APIs for MPI

We consider the future research directions for the two research APIs described in this paper.

8.0.1 MPI++

The design of MPI++ discussed conservative choices about the use of C++ features. Specifically, in our effort to retain a large measure of compatibility with the C interface, certain interfaces remain C-like. However, the class structure of C++ was exploited in some cases.

What is absent here, largely because MPI objects are already reference-counted by virtue of the application-programmer-interface semantics, is discussion of nested classes, inheritance, and the handle/body (or envelope/letter) idioms discussed by Coplien [8]. The MPI++ objects that implement opaque MPI objects are relatively simple because the C objects they encapsulate are already quite easy to manage. However, because the C binding has neither hooks for inheritance, nor an external interface to support tight binding of the C++ interface, certain concessions were inevitable.

For example, it is not possible to layer certain calls involving arrays of request objects on top of the equivalent MPI calls (*e.g.*, `MPI_Wait` and `MPI_Waitall`). Specifically, one cannot pass an array of C++ requests to the C routine, because the objects differ in content and format. For this reason, it is tempting not to use some of the built-in MPI functionality that works on arrays of C objects, but rather to use more primitive MPI functions, and do the array versions in the MPI++ implementation. However, this could dramatically affect the performance and/or deadlock properties of the C++ version. Our design requirements forbid so large a deviation from the original underlying implementation.

A workable solution to this problem is to consider a ground-up implementation of MPI in C++, basing the system on an appropriate abstract network architecture, as is commonly done. Considerable improvements would potentially result if the entire system were C++, rather than just the uppermost layers, and some of these could lead to improved functionality as well as higher performance. Such an implementation is an ideal next step for the MPI++ project.

Application Experience Relatively small quantities of code have been written in MPI++ as compared to MPI's C and F77 bindings, but substantial code now exists. Hence, we lack the breadth of feedback that such experience would provide about the efficacy of features we have provided, and the usefulness of the extensibility of the classes we have defined. For example, we have not explicitly provided for multiple inheritance based on our classes, although we have followed Coplien's Inheritance Canonical Form for simple inheritance [8]. We expect to get additional feedback from application programmers, which could lead to significant additions or changes in functionality. Furthermore, other approaches to C++-based message passing are likely to give us impetus for change and improvement.

At the outset of this work, we were already convinced that a class library based on MPI could provide a useful parallel C++ environment. What remains to be studied is whether an inherently parallel compiler-based environment, with commands to support and manage parallelism, can provide things that C++, as extensible as it is, cannot. We look forward to such a comprehensive study, as we believe this will help drive further research into the most fruitful directions for parallel C++ systems. Furthermore, we look to significant application experience as a guide toward what MPI++ should and should not provide to support libraries and applications.

At present, MPI++ has been transfromed to offer the standardized C++ language bindings for the MPI-1 subset of MPI-2[12]

8.0.2 OOMPI

The next major release of OOMPI (1.1) will be built upon the C++ MPI bindings, and could utlilize the newest MPI++ release for that purpose. This will enable full use of `const` semantics. While this semantic change in the OOMPI interface may cause problems in terms of backwards compatibility, correctly written OOMPI programs should not be adversely affected. Building OOMPI on the C++ MPI bindings will simplify the underlying implementation of OOMPI. This will reduce the overhead of a typical OOMPI call, and therefore result in higher performance. Some additional functionality will probably be offered as well. For example, attribute caching, OOMPI to MPI casting operators, and enhanced datatype constructors are potential new features.

Finally, OOMPI 2.0 is projected to include support for all of MPI-2, once MPI-2 implementations become prevalent.

[12]This is available online [13] code and examples available at http://www.erc.msstate.edu/labs/hpcl/mpi++.

9 Conclusions

In this paper, we presented two object-oriented class libraries for supplementing the MPI system, in support of C++. The features and limitations of these systems were presented, as was an after-the-fact object-oriented analysis of the application programmer interface and semantics of MPI-1. We compared the two class libraries to the C++ interface accepted by the MPI Forum. Furthermore, we noted that both class libraries have formed the basis for support of this interface in various forms of MPI implementations. The main lessons here are that C++ needed a class library to work comfortably with the MPI application programmer interface, that several levels of abstraction make sense, and that these still may be particularly of interest to users in view of the spartan support introduced by the MPI Forum itself.

The lessons learned about the objects and interactions of MPI, obtained herein, offer insight into the forward design of extensions and future interfaces analogous to MPI, representing an important contribution to further efforts in such standard-oriented efforts. Ideas from this work have had a positive impact on the real-time message passing interface standard (MPI/RT), which has exploited object-oriented analysis and design from the outset.

Acknowledgments

The authors acknowledge helpful input on this paper from Shane Hebert of Mississippi State University, Nathan Doss (formerly of MSU, currently with Lockheed-Martin Government Electrical Systems), Mike McNally, and Jeremy Siek (University of Notre Dame), all for their intellectual contributions. We also thank Bill Saphir (LBL/NERSC), and Jim Cownie (Dolphinics) for their helpful input.

References

[1] Jean-Marc Adamo. Arch, An Object-Oriented Library for Asynchronous and Loosely Synchronous System Programming. Technical Report CTC95TR228, Cornell Theory Center, Ithaca, NY 14853-3801, January 1996.

[2] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[3] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[4] Grady Booch. *Object–Oriented Analysis and Design with Applications.* Addison–Wesley, Santa Clara, California, second edition, 1994.

[5] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern–Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, Chichester, New York, 1996.

[7] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Journal of Parallel Computing*, 1994. to appear (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).

[8] J.O. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison–Wesley, 1992.

[9] Olivier Coulaud and Eric Dillon. Para++: C++ bindings for message passing libraries user guide. Tech. report, INRIA, 1995.

[10] Olivier Coulaud and Eric Dillon. *Para++: C++ Bindings for Message Passing Libraries User Guide*. Institut National de Recherche en Informatique et en Automatique, version 1.1 edition, April 1996.

[11] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–75, April 1993.

[12] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.

[13] Stephen H. Edwards. Streams: A Pattern for "Pull–Driven" Processing. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison–Wesley, Menlo Park, California, 1995.

[14] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[15] Message Passing Interface Forum. MPI-2: Extensions to the message passing interface. Technical Report Technical Report No. CS-96-XXX, University of Tennessee, November 1996. Available on **netlib**.

[16] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison–Wesley, Menlo Park, California, 1997.

[17] Erich Gamma, Richard Helm, Ralpha Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

[18] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: A Parallel Virtual Machine*. Scientific and Engineering Computation Series. MIT Press, 1994.

[19] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22:789—828, 1996.

[21] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

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

[23] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990.

[24] Parasoft Corporation, Pasadena, CA. *Express User's Guide*, version 3.2.5 edition, 1992.

[25] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[26] Douglas C. Schmidt and Charles D. Cranor. Half–Sync/Half–Asynch: An Architectural Pattern for Efficient and Well-Structured Concurrent I/O. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*. Addison–Wesley, Menlo Park, California, 1996.

[27] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990.

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

[29] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P . Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, April 1994. (Invited Paper, in Special Issue on Message Passing).

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

[31] A. Weinand, E. Gamma, and R. Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2), 1989.

[32] Walter Zimmer. Relationships between design patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison–Wesley, Menlo Park, California, 1995.