Explicit Parallel Programming in C++ based on the Message-Passing Interface (MPI)*

Anthony Skjellum, Ziyang Lu,
Purushotham V. Bangalore, Nathan Doss
Mississippi State University
Department of Computer Science
and
NSF Engineering Research Center
for
Computational Field Simulation
Mississippi State, MS 39762

September 15, 1995

1 Introduction

Explicit parallel programming using the Message Passing Interface (MPI), a de facto standard created by the MPI Forum [15], is quickly becoming the strategy of choice for performance-portable parallel application programming on multicomputers and networks of workstations, so it is inevitably of interest to C++ programmers who use such systems. MPI programming is currently undertaken in C and/or Fortran-77, via the language bindings defined by the MPI Forum [15]. While the committee deferred the job of defining a C++ binding for MPI to MPI-2 [16], it is already possible to develop parallel programs in C++ using MPI, with the added help of one of several support libraries [2, 6, 13]. These systems all strive to enable immediate C++ programming based on MPI. The first such enabling system, MPI++, is the focus of this chapter.

MPI++ was an early effort on our part to let us leverage MPI while programming in C++. Here this system is, to a large extent, our vehicle to illustrate the value added of C++ in a message passing environment and, conversely, the

*To appear in *Parallel Programming Using C++*, Gregory V. Wilson, Editor. MIT Press, 1996.

value of MPI towards parallel programming with C++. We will be describing a performance-conscious alternative to exploit parallelism with C++ without the benefit of a portable, mature compiler environment suitable for a network of workstations or massively parallel computer. We emphasize performance-portability — good performance and good portability at the same time — and good design issues throughout, and that will put constraints on how eagerly we exploit certain features of C++ when creating our parallel environment on top of MPI.

1.1 MPI Overview

As a computational paradigm, message passing is widely used in parallel programming. During the development of massively parallel and cluster environments for parallel processing during the 1980's, there was neither sufficient portability of codes nor an agreed-upon notation for expressing such programs (syntax and semantics). To overcome this problem, many portability libraries were invented, such as P4 [3], Zipcode [18], Chameleon [12], PARMACS [4], and PVM [10]. Importantly, these systems all differ in the kind, flexibility and quality of services they can provide, as a function of their emphasis on tightly coupled or loosely coupled machines, together with their diverse syntactical limitations and operational semantics. That meant that a standard could not be created by merely "choosing" one of the existing vendor or portability systems available at that time. However, by mid-1992, it was clearly time to define a standard for message passing system that would incorporate and improve the best features of those systems and existing practice [14]. MPI is such a standard, developed in a de facto working group called the MPI Forum, with input and participation from industry, national laboratories, and universities [15]. The authors of all the key vendor and portability systems participated in defining the standard.

MPI provides many features that support writing portable, efficient, scalable parallel programs. Because of space limitations, we must forego many of the details of its syntax, semantics, and practical assumptions, but we summarize these issues briefly below. You might also like to refer to [11], which is an accessible introduction to MPI; the formal standards document is also worth reviewing [15]. Together with Web tutorials mentioned in Section 1.3, you should have no trouble getting a detailed appreciation of MPI. In this chapter, we concentrate mainly on some key concepts of MPI that are directly relevant to our design of MPI++, and to the scope of the parallel models achievable with it. A basic appreciation for these concepts is critical, as is an appreciation for MPI's persistent objects. That is so because MPI objects, such as "communicators" and "groups," provide clean, consistent opaque objects that fit naturally into the object-oriented world. They further dictate that essentially all MPI implementations are object-based in many respects, leading to some simplifications and some additional complexities for MPI++.

1.1.1 Summary of MPI Features

MPI supports both point-to-point and collective message passing among communicating sequential processes. The MPI standard defines the syntax and semantics for MPI functionality, and it also defines a C and Fortran-77 language binding for the standard. The following classes of operations are supported:

- Point-to-point transmission with various "modes" (blocking, non-blocking, synchronous, buffered, and several combinations thereof),
- Collective communication processes specified by groups (barrier, reductions, broadcasts, gathers, scatters, and scans),
- Constructors for formulating communication structures (define subgroups, build new communicators, duplicate existing communicators),
- Mechanisms for single group (data parallel), and dual-group (dataflow or bi-partite-style) communication structures,
- Virtual topologies that allow processes to be named and manipulated in a scheme more suited to application use.

In particular, the following major concepts are defined by MPI:

group An ordered collection of processes; processes in a group have unique ranks: $0...N-1$, for N processes. All communication has a group scope. Groups are mainly used within communicators (see below).

context An abstract mechanism by which processes in a group separate their communication from other on-going communication in the system. Constructors for communication provide this context information within communicator structures.

rank The name of a process relative to a group. The most fundamental names of processes in MPI are ranks in `MPI_COMM_WORLD`'s group.

communicator The opaque object that is used to select the group and the context of communication; they are used for both point-to-point communication (where group ranks become the names of senders and receivers), and collective communication (where group membership defines who participates). Intracommunicators work on a single group; intercommunicators work on two groups. In MPI-1, intercommunicators have limited collective operations, but this is being remedied in MPI-2.

data type An opaque object that describes the kind of information being transferred (such as integers). Predefined and user-defined data types allow MPI to encapsulate heterogeneous conversions without user intervention, while providing a higher-level interface for gathering and scattering of non-contiguous data being transferred.

request object Any outstanding, non-blocking operation (*e.g.*, non-blocking send or receive) gives the user an opaque handle for later inquiry about that outstanding operation. At completion of an operation (or if there should be an error), a request can be transformed into a status object, which gives the user further information about the completed operation.

status object A status object is a non-opaque object that the user can peek into to see what happened when one of a large number of MPI operations completes, most usually for non-blocking operations.

For definiteness, Table 1 lists a usable subset of MPI calls. A total of 129 calls are provided by the standard, but any given application will utilize a much smaller set of features. MPI has a few key concepts implemented with an appropriate number of functions. This renders it easy to learn.

MPI provides an initial communicator (`MPI_COMM_WORLD`) that contains all processes, while MPI-2 [16] extends these notions to support expandable "worlds." The user can create new communicators using constructors provided that split or merge existing communicators to form new communication space.

Finally, MPI defines a rule concerning progress requirements for all implementations. While this is a point of contention with implementors, the guarantee of progress sets MPI apart from other systems in terms of its high expectations for functioning properly on diverse systems.

1.1.2 Parallel Models Supported

Since MPI++ is based on MPI, it is important to note that several computational models are supported readily by MPI, and hence by MPI++. In particular, the following models are supported:

Data Parallel Each MPI communicator supports data parallel as well as unstructured communication among group members. This is the simplest way to program MPI. Lexical scoping of messages for nested parallel calls is also supported, as noted in Figure 1, principally via `MPI_COMM_DUP`. Furthermore, in concert with task parallelism mentioned next, DAG-type computational structures can be constructed.

Task Parallel Communicators can be split into K disjoint new communicators with a single call (where K is at most the number of processes in the original communicator); this is done using `MPI_COMM_SPLIT`. Successive calls to this function can be done to create complex, hierarchical, and overlapping communication spaces. Figure 2 demonstrates a simple instance of task parallel programming, based on the notion of splitting the processes into disjoint groups, followed by independent computation, and concluded by communication and/or computation of the entire group.

Extremely basic subset:	
`MPI_INIT`	Initialize MPI
`MPI_COMM_SIZE`	Find out how many processes there are
`MPI_COMM_RANK`	Find out which process rank I am
`MPI_SEND`	Send a message (blocking)
`MPI_RECV`	Receive a message (blocking)
`MPI_FINALIZE`	Terminate MPI
Non-blocking point-to-point and related:	
`MPI_ISEND`	Send a message (non-blocking)
`MPI_IRECV`	Receive a message (non-blocking)
`MPI_WAIT`	Wait on pending message
`MPI_GETCOUNT`	Get size of received message
Group-scoped collective communication:	
`MPI_BCAST`	One-to-all broadcast
`MPI_REDUCE`	All-to-one reduction operation
`MPI_ALLREDUCE`	All-to-all reduction operation
Communicator constructors/destructor:	
`MPI_COMM_SPLIT`	Make K new communicators from one
`MPI_COMM_DUP`	Duplicate a communicator
`MPI_COMM_FREE`	Communicator deletion operation
Some Data motion operations:	
`MPI_GATHER`	Gather data into one process
`MPI_SCATTER`	Scatter data across processes
Support for timing:	
`MPI_WTIME`	Timing routine (relative time)
`MPI_WTICK`	Timing routine (clock tick size)

Table 1: A usable subset of MPI Functions. The communication routines work relative to communicators, which further defines the group of participating processes. Ranks within groups define names for send/receive primitives.

Medium-grain Dataflow Model Intercommunicators can be constructed that represent two groups, and support communication strictly *between* the groups. The full-duplex communication supported can be used to model dataflow computations, as well as symmetric communication between two parallel groups (*e.g.*, parallel client-server communication). In Figure 3, a simple, two group dataflow computation is sketched, similar in nature to the task parallel computation shown in Figure 2, but this time incorporating inter-group communication, as well as a dependency on the output of one group ($G1$) on the computation of the other ($G2$).

While, individually, each model is quite straightforward, MPI provides no a priori limitations on the combination of the above forms of computation, enabling the construction of rather complex applications. Furthermore, because communicators provide safe communication space, decisions made about sending messages deep within a computation do not affect the correctness of other parts of an application, assuming communicators are used in a consistent fashion. MPI++ inherits these desirable features from MPI.

One noteworthy limitation of MPI, to be relaxed in MPI-2, is the requirement that all processes be created at the outset of the computation. Relaxing this requirement will lead to multiple worlds, to be connected by intercommunicators and/or intracommunicators, depending on the way new processes join the computation, and based on design choices made by the application programmer.

1.2 MPI++ Origins and Overview

We stated at the outset that the C++ language binding to MPI was deferred to MPI-2. We were nonetheless certain that such language support would benefit programmers who wanted to do parallel C++ and who were also interested in MPI's performance and portability. Practical programmers working on real applications are, however, generally unwilling to accept big degradations of performance for nice programming models, but will accept modest tradeoffs of peak performance for worthwhile software engineering benefits; this guided our mindset on what to develop. Specifically, a C++ interface would ideally allow programmers to write message-passing code in an object-oriented programming style, without huge drop off in bandwidth and large added latency. This idea of taking advantage of both MPI and Object-oriented programming led us to MPI++, which started in early 1994 [2]. The application architecture we sought to support is shown in Figure 4.

Another goal of our C++ interface to MPI was to provide an extensible, portable, efficient, heterogeneous-capable object-oriented message-passing library that could be used to build scalable, object-oriented numerical libraries. This is motivated by the desire to convert our existing numerical libraries in the *Multicomputer Toolbox* [8, 17] that were originally written in C using Zipcode; Zipcode itself provided many object-oriented features to message passing,

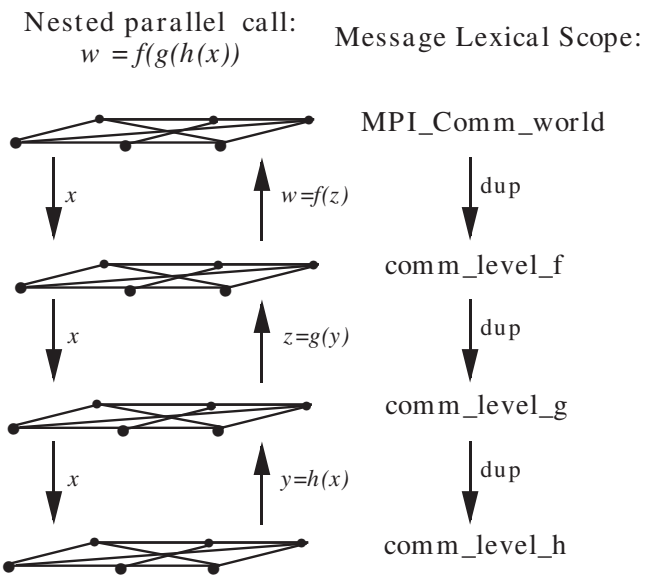

Figure 1: The figure illustrates the use of MPI in a canonical fashion: nested, parallel calls with the same group participants can be handled easily, and each nested level gets its own safe communication space using `MPI_COMM_DUP`. In particular, for each parallel function f, g, and h, needed messages are lexically scoped within a single level of the parallel computation (message move only horizontally in this conceptual figure).

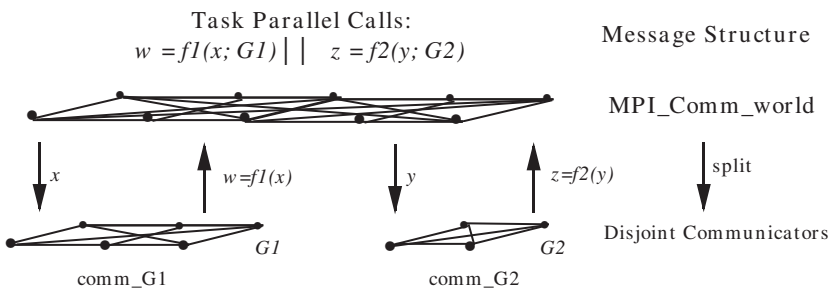

Figure 2: Task parallel programming is supported by MPI and MPI++ through the use of multiple communicators. In the figure, `MPI_COMM_WORLD` is split into two disjoint groups, each of which can then work independently on a parallel task.

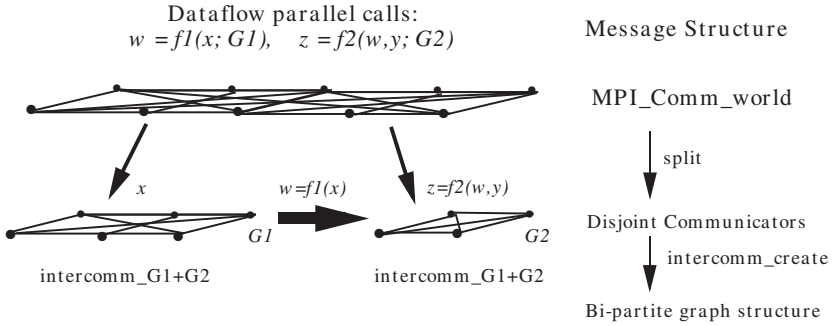

Figure 3: The use of intercommunicators supports full-duplex, group-to-group communication; this can be used to do dataflow computations of a medium-grain nature. The figure sketches a dataflow computation for two groups. Any pair of groups can be connected in this way, permitting longer pipelines, as well as structures that include feedback loops.

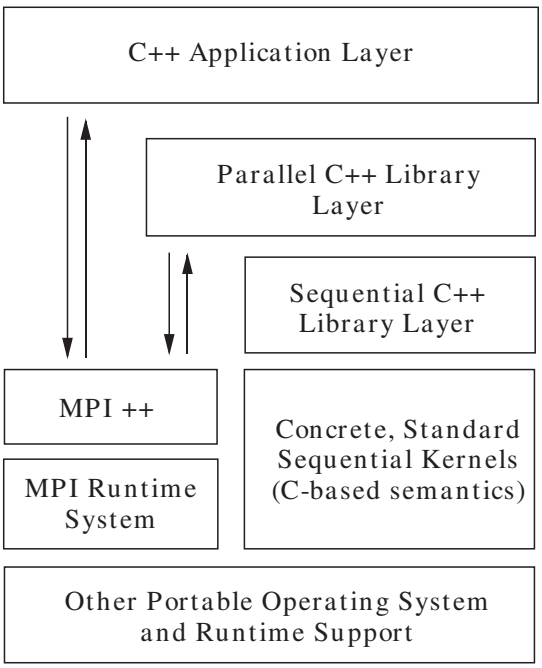

Figure 4: At a conceptual level, this is the parallel application architecture currently supported by MPI++. It would ideally contain both standardized as well as application-specific reusable software written in C++.

but was constrained by its C interface. MPI++ is our first step to provide infrastructure for converting the Toolbox libraries to C++. Moreover, MPI++ should help make any parallel libraries reliable and easy to use, as it may help encapsulate and hide details usually involved in building libraries.

MPI++ should also promote reusability and extensibility, provide better encapsulation and better support for managing complexity. In particular, MPI++ should ease the job of writing parallel libraries, and development of parallel applications as well. Given these goals, we also have a starting hypothesis that guided our design: *a good class library for MPI goes a long way to providing a parallel C++ programming environment.* Working to offer supporting arguments and/or counterexamples for this hypothesis provides a defining theme for our evaluations of strengths and weaknesses of MPI++ in what follows, and, we argue, for your own.

MPI++ has evolved since we started it in 1994. It will continue to evolve since technologies in sequential C++ compilers are becoming mature (such as better support for templates), plus MPI itself will expand, as MPI-2 is still an on-going effort, meant to continue at least through the end of 1996.

1.3 Net Resources for this Chapter

The following is a brief description about how to obtain materials related to this chapter on the Internet.

- The MPI++ source code is available through the WWW at:

  `http://www.erc.msstate.edu/mpi/mpi++/`.

  The polygon overlay examples used in this chapter are also available at this site.

- MPICH, the model implementation of MPI developed by Argonne National Laboratory and Mississippi State University, is available by anonymous `ftp` from `info.mcs.anl.gov`. The '`README`' file in directory '`pub/mpi`' describes how to fetch and install the most recent version of the model implementation.

- The MPI Standard document [16] is available by anonymous `ftp` from '`pub/mpi/mpi-report.ps.Z`' at `info.mcs.anl.gov`. It is also available on the WWW at:

  `http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html`.

- General information on MPI is available on the WWW at:

  `http://www.erc.msstate.edu/mpi/`.

  A "frequently asked questions" (FAQ) is available at:

`http://www.erc.msstate.edu/mpi/mpi-faq.html`.

There is also a newsgroup, `comp.parallel.mpi`, where all aspects of MPI are discussed.

1.4 Acknowledgments

Support has been provided by the U.S. Department of Energy under Contract DE-FC04-94AL9892. Additional funding for this work is acknowledged under ARPA Order D350, Contract through Rome Laboratories, 1995. This research has also been supported in part by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University.

We also would like to thank the Maui High Performance Computing Center where all the performance tests for the polygon overlay application were conducted.

2 Design and Implementation Aspects of MPI++

We start this section by summarizing the list of requirements we set out at the beginning of this project, and that has guided our design and implementation throughout:

- The MPI++ interface should be consistent with the C interface.
- Performance-portability should be comparable to the C interface. That is, the C++ features should be lightweight whenever the corresponding C function is lightweight.
- MPI++ should exhibit good object-oriented features.

In the following subsections, we will consider each of these requirements. In particular, we shall discuss the rationale behind these requirements, their influence on our choices of design decisions as well as the way we carried out the implementation. By way of contrast, in Section 5 we consider the implications of altering these requirements for future C++ design based on MPI.

2.1 The MPI++ Interface

MPI's language-independent semantics are not subject to rapid change, because of standardization. It is thus attractive (as well as prudent) to follow the MPI standard carefully when we design and implement a C++ interface for it, even though this is not to be an official language binding. Moreover, MPI does have a C language binding, so many potential users of MPI++ already have a working knowledge of its C interface. Ideally, it should take such users minimal effort

to migrate from the C interface to MPI++. These considerations strongly suggest that the MPI++ interface be consistent with the C interface, by which we mean both syntactically and semantically. Of course, since C and C++ are two different languages, it is impossible (and even unnecessary) to insist on exact syntactic correspondences. But we did choose to have an obvious mapping from one interface to another, meaning a user who knows the C interface can easily recognize the counterparts of C++ interface. Semantic consistency requires that the counterparts in the two interfaces must perform the same function without unexpected side effects (including performance). This prevents undesirable erroneous uses or unexpected performance hits when one switches from one interface to the other.

The current MPI++ interface has taken the above requirements into account and is conceptually consistent with the standardized C interface. A good way to appreciate their relationship is with some C and C++ examples.

2.1.1 Simple Example #1

Here is the C version of an SPMD program adapted from the MPI document [15].

```
#include <mpi.h>
#define MAX_MESSAGE_SIZE  20
#define MESSAGE_TAG       99
main (int argc, char **argv)
{
  char       message[MAX_MESSAGE_SIZE];
  int        rank;
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  MPI_Comm_size ( MPI_COMM_WORLD, &size );
  if (rank == 0) {
    strcpy(message,"Hello, there");
    MPI_Send(message,strlen(message)+1,MPI_CHAR,size-1,
             MESSAGE_TAG,MPI_COMM_WORLD);
  }
  else if (rank == (size-1)){
    MPI_Recv(message,MAX_MESSAGE_SIZE,MPI_CHAR,0,
             MESSAGE_TAG,MPI_COMM_WORLD,&status);
    printf("received :% s:\ n'', message);
  }
  MPI_Finalize();
}
```

As illustrated by this program, every MPI program must initialize the MPI environment by calling `MPI_INIT` before calling any other MPI functions. A

call MPI_FINALIZE is made at the end of the program to clean up the MPI environment. MPI_COMM_SIZE is used to determine the number of processes in the communicator. MPI_COMM_RANK is used to determine the position or rank of the calling process in the communicator. Process zero (rank = 0) sends a message to the highest-numbered process (rank = size-1). All other processes simply fall through the if-else and terminate.
Here's the MPI++ version:

```
#include <mpi++.h>
const int MAX_MESSAGE_SIZE = 20;
const int MESSAGE_TAG      = 99;
main (int argc, char **argv)
{
  char       message[MAX_MESSAGE_SIZE];
  int        rank;
  MPIX_Status status;

  MPIX_COMM_WORLD.Init (&argc, &argv);
  MPIX_COMM_WORLD.Rank (&rank);
  MPIX_COMM_WORLD.Size (&size);
  if (rank == 0) {
    strcpy(message,"Hello, there");
    MPIX_COMM_WORLD.Send(message,strlen(message)+1,MPIX_CHAR,
                         size-1,MESSAGE_TAG);
  }
  else if (rank == (size-1)) {
    MPIX_COMM_WORLD.Recv(message,MAX_MESSAGE_SIZE,MPIX_CHAR,
                         0,MESSAGE_TAG,&status);
    cout << ``received'' << message << endl;
  }
  MPIX_COMM_WORLD.Finalize();
  return (0);
}
```

The first thing to note is that the statement `#include <mpi.h>`, in the C version, has been replaced by `#include <mpi++.h>`. 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 with the C interface since MPI++ was implemented on top of the C interface [1]. However, the biggest difference between the two versions is that the communicator (MPIX_COMM_WORLD) is an object with methods (member functions) in the C++ version. For instance, the function calls in the C version:

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

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

are changed into method invocations of object MPIX_COMM_WORLD:

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

Likewise, the send and receive calls that, in the C code, take the communicator MPI_COMM_WORLD:

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

are also changed to methods of object MPIX_COMM_WORLD:

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

The other arguments remain the same. The two "book end" calls in the C version

```
MPI_Init (&argc, &argv);
MPI_Finalize();
```

are replaced by the corresponding methods of object MPIX_COMM_WORLD in C++:

```
MPIX_COMM_WORLD.Init (&argc, &argv);
MPIX_COMM_WORLD.Finalize();
```

More generally speaking, in the C++ version, most MPI functions are available as methods of a class. MPIX_COMM_WORLD is an instance of the MPIX_Comm_world class, and as such, has many of the MPI functions that in C take a communicator as an argument. Other than that, these two bindings are clearly similar. One way to describe the relationship is to say that we have moved the communicator argument to the front of the function call and put a "." instead of "MPI_" or "MPI_Comm." While this is not a technical description, it is accurate and does represent how simple MPI code in can be changed into MPI++ code. (But this is not the end of the story. With the class hierarchy described in the next subsection, as well as examples in Section 3, we will show you the added value and power of the MPI++ interface).

The above sample program also contains some examples that show the choices we made to keep a consistent interface. For instance, it is possible to pass arguments by reference in C++; however, we've chosen to keep the calling conventions the same as in the C interface and have you pass the arguments by using the address operator instead of letting C++ automate this for you.

```
MPIX_COMM_WORLD.Rank(rank);   // oops, bad.
MPIX_COMM_WORLD.Rank(&rank);  // good.
```

We could have made either one of the above work, but we chose to only support the second. We will discuss this and other options further in Section 5.

2.2 Object-oriented Design of MPI++

Different object-oriented designs lead to distinct "flavors" of C++ software. Here we discuss MPI++'s design in more detail.

2.2.1 The Class Hierarchy

As we discussed in Section 1.1, the MPI specification is object based, therefore it is relatively easy to identify the major classes for MPI++. The class hierarchy can take one of the two forms: single-rooted or forest [19]. It proved possible for us to use either approach in the design of MPI++. However, there are trade-offs between these two choices. For instance, the single-rooted approach (in which all objects are ultimately derived from an abstract superclass) results in a more uniform and sophisticated system. Yet, this style of design is most suited for a second phase of object-oriented design where further departure from the C interface syntax and semantics would be indicated. On the other hand the forest approach (in which a number of superclasses support the user-accessible objects) fits naturally into the MPI specification, and thus the resulting system is more likely to be consistent with the C interface, and is also easier to build as a first-cut object oriented design.

2.2.2 The Communicator-based Classes

The communicator-based classes constitute the backbone of the MPI++ architecture, and its hierarchy is illustrated in Figure 5. `MPIX_Comm` is the base class which contains all the functions that are common to both intra- and inter-communicators, such as point-to-point communication and accessor functions. Its interface is sketched out in Program 1.1.

```
// Declarations
class MPIX_Comm;
class MPIX_Comm_world;
extern MPIX_Comm         MPIX_COMM_NULL;
extern MPIX_Comm_world   MPIX_COMM_WORLD;
class MPIX_Comm {
public:
    // Constructor ----------------------------------------//
    MPIX_Comm( );
    // Destructor -----------------------------------------//
    virtual ~ MPIX_Comm( );
    // Free a constructor ---------------------------------//
    virtual int Free( );
    // Environment ----------------------------------------//
    virtual int Abort( int );
    virtual int Errhandler_set( MPI_Errhandler& );
    virtual int Errhandler_get( MPI_Errhandler* );
    // Accessors ------------------------------------------//
```

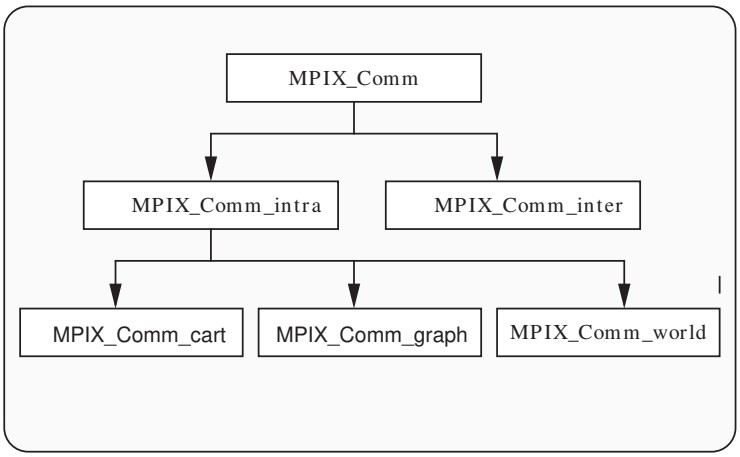

Figure 5: Hierarchy of communicator-based classes

```
    virtual int Size( int * );
    virtual int Rank( int * );
    .
    .
    .
    // Initializers ----------------------------------------//
    virtual int Dup( MPIX_Comm* );
    // 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 --------------------------------//
    // = operator makes a *copy*, not a duplication
    virtual MPIX_Comm& operator=(const MPIX_Comm& old_comm)
    virtual int operator==(const MPIX_Comm&)
    virtual int operator!=(const MPIX_Comm&)
private:
    .
    .
    .
```

```
}
```

Program 1.1: Interface of the `MPIX_Comm` class

The `MPIX_Comm_intra` (intra-communicators) class is derived from `MPIX_Comm` and contains collective communication and topology functions as well as various other functions that do not apply to inter-communicators. A sketch of its interface is listed in Program 1.2.

```
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*);
    ⋮
};
```

Program 1.2: Interface of the `MPIX_Comm_intra` class

The `MPIX_Comm_inter` (inter-communicators) class is also derived from the `MPIX_Comm` class, and contains only those functions peculiar to itself. Program 1.3 gives a sketch of its interface.

```
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*);
};
```

Program 1.3: Interface of the `MPIX_Comm_inter` class

While most of these methods are straightforward to use, we point out a few things with which the user needs to be careful. For instance, we decided that the destructor for the `MPIX_Comm` class should not free the communication resources associated with the communicator, thus all instances of `MPIX_Comm` must be explicitly freed as in C with `MPIX_Comm::Free()`.

```
Swap_buf ( int *buf, MPI_Comm_intra& old_comm )
{
    MPIX_Comm_intra  comm ( old_comm ); // actually creates a new communicator

    comm.Send ( buf, 1, MPI_INT, dest,   tag );
    comm.Recv ( buf, 1, MPI_INT, source, tag );

    comm.Free();                // must be explicitly freed

    // "comm" goes out of scope so its destructor is called
}
```

Users must be careful to distinguish between using the overloaded "=" operator and instantiation, via the communicator's copy constructor:

```
  // communicator instantiation calls
  // MPIX_Comm_intra::MPIX_Comm_intra(const MPIX_Comm&)
  // constructor which creates a duplicate of comm1
  MPIX_Comm_intra c1 = comm1; // parallel call
```

By way of contrast, `operator=` is a local call:

```
  // 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
```

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()`. This follows the logic that C operators, extended to C++ objects, should not have unexpected side effects.

```
    MPIX_Comm_intra comm ( MPI_COMM_WORLD );  // creates a duplicate
                                              // of MPI_COMM_WORLD
    // Default initialization, setting to MPI_COMM_NULL
    MPIX_Comm_intra world;

    world = comm; // world now points to the same communicator
                  // as "comm" does

    world.Free(); // ok
    comm.Free();  // erroneous
```

In principle, the above restrictions can be avoided by using techniques such as reference counting [5]. But the real issue is the potential for deadlock. Should = require loosely synchronous invocation over the entire group involved in the communication.

Figure 5 also shows that three subclasses are derived from MPIX_Comm_intra, namely, MPIX_Comm_world, MPIX_Comm_cart, and MPIX_Comm_graph. MPIX_Comm_world is special in that there will always be one and only one instance of this class during an MPI program. This instance will be called MPI_COMM_WORLD. To prevent users from creating additional instances of this class, MPIX_Comm_world is designed and implemented as a singleton [9]. This class has two methods that are not in the MPIX_Comm_intra class that are used to manipulate the MPI environment. These methods are listed below.

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

The other two subclasses derived from MPIX_Comm_intra provide support for virtual topologies. Virtual topologies provide a machine-independent naming abstraction to describe communication operations in terms that are natural to an application, and thus are very helpful in building parallel libraries, specifically the support for virtual names for processes in a communicator's process group. MPIX_Comm_cart supports cartesian topologies which are decompositions into rectangular coordinates. MPIX_Comm_graph supports general graph topologies, where processes become nodes of a topology graph.

2.2.3 The Group Class

Now we turn to another class MPIX_Group. Program 1.4 shows a sketch of the interface of the MPIX_Group class.

```
// Declarations -------------------------------------------//
class  MPIX_Group;
extern MPIX_Group        MPIX_GROUP_EMPTY;
extern MPIX_Group        MPIX_GROUP_NULL;
class MPIX_Group {
public:
    // Constructor --------------------------------------//
    MPIX_Group (void);
    MPIX_Group (const MPIX_Group& old_group);
    // Destructor ---------------------------------------//
    virtual ~MPIX_Group (void);
    // Free a group -------------------------------------//
    virtual int Free(void);
    // Group accessors ----------------------------------//
    virtual int Size(int *);
    ⋮
    // Group manipulation -------------------------------//
```

```
    virtual int Union(const MPIX_Group&, MPIX_Group*);
    .
    .
    .
    // Assignment ------------------------------------------//
    virtual MPIX_Group& operator=  (const 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;
    .
    .
    .
};
```

Program 1.4: Interface of the `MPIX_Group` class

Most of the `MPI_Group` methods are straightforward. `MPIX_Group` objects are not explicitly used in communication, their only use in MPI is for describing group members when building new communicators using the `MPIX_Comm::Create` method. As such, they are not as important as the communicator classes in the majority of MPI++ programs, though we expect their wide use in MPI-2. There are two constructors:

```
MPIX_Group::MPIX_Group( ) //Creates an empty group

MPIX_Group::MPIX_Group(MPIX_Group& ) //Creates an exact copy
```

2.2.4 The Datatype Class

In addition to the basic datatypes such as integer, float, char, and so on, MPI provides functions for building derived data types such as vector, indexed, and struct datatypes. In general, new datatypes are built recursively from previously

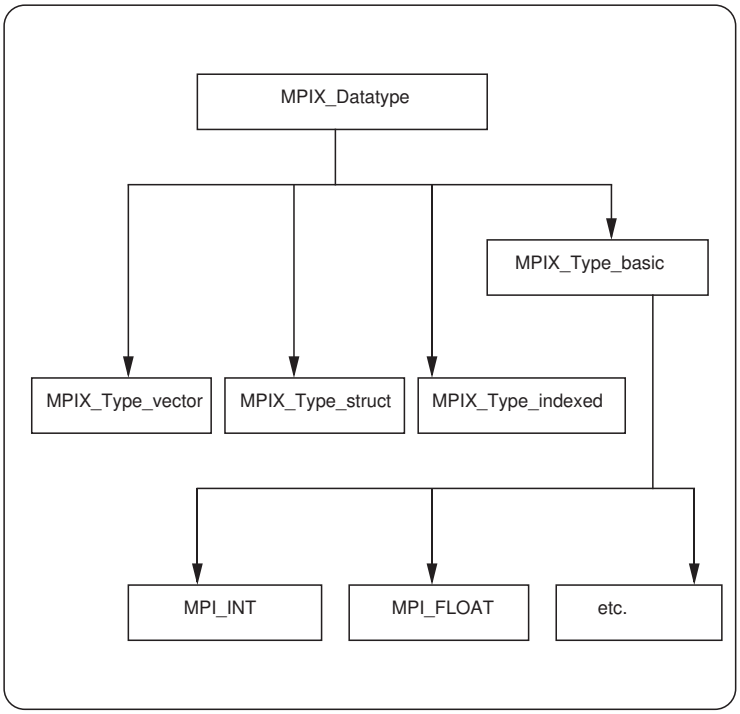

Figure 6: Hierarchy of datatype-based classes

built datatypes. Figure 6 shows our interpretation of the hierarchy of datatype-based classes for MPI++.

Datatypes used in the MPI C binding basically fit into this picture well. These are abstract objects. So in the present implementation of MPI++; we have retained them without further abstraction.

2.2.5 The Handler Classes

The four remaining classes in the MPI++ class hierarchy have to do with 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 instance, the C function

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

becomes

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

Program 1.5 shows a sketch of the interface for `MPIX_Request`.

```
// Declarations ---------------------------------------------//
class  MPIX_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;
    .
    .
};
```

Program 1.5: Interface of the `MPIX_Request` class

And the interface for `MPIX_Status` is illustrated in Program 1.6

```
// Declarations ---------------------------------------------//
class  MPIX_Status;
class MPIX_Status {
public:
    // Constructor ----------------------------------------//
    MPIX_Status ( );
    // Destructor -----------------------------------------//
    virtual ~MPIX_Status ( );
    // Status accessors -----------------------------------//
    virtual int MPI_TAG( );
    virtual int MPI_SOURCE( );
    virtual int Test_cancelled( int * );
        .
        .
```

```
private:
    MPI_Status status;
        ⋮
};
```

Program 1.6: Interface of the `MPIX_Status` class

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`.

2.2.6 Comments on Design

From the above discussion of the MPI++ class hierarchy, one can see that it supports most of the major functionality that MPI provides. The remaining features currently missing are mostly of an auxiliary nature and can be added easily. Moreover, it has and/or uses the following features and provides certain benefits from each:

1. Inheritance (white box reuse). In particular, the communicator-based class hierarchy gives the user much more flexibility and promotes reusability and extensibility. For example, if a particular application requires a new 2D Cartesian topology of dimension $P \times Q$, then the user can create one by inheritance and customize it with minimal work. A sample interface might look like this:

```
class MPIX_Grid2d : public MPIX_Comm_intra {
public:
  // Row and column communicators
  MPIX_Comm_intra Row, Column;
   // Constructor
  MPIX_Grid2d(void);
   // Destructor
  ~ MPIX_Grid2d(void);
   // Free grid
  int Free(void);
   // Initialize grid
  int Init(MPIX_Comm_intra& comm_in, int P, int Q);
   // Duplicate grid
  int Dup(MPIX_Grid& grid_out);
   // 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_;
  itn p_, q_;
};
```

where the row (column, resp.) communicator could consist of all the processes in the same process row (column, resp.) of the grid and can be used for communications local to the process row (column, resp.). Point-to-point functions can be overloaded to accept grid-specific process names such as coordinate (5,3) instead of the standard process rank in group. Such functionality had to be hard-wired into our previous system Zipcode. Other notational conveniences are possible too.

2. Aggregation (black box reuse). The aggregation mechanism is used extensively, but only implicitly in our design. This is so because our design takes advantage of the underlying MPI structure. The resulting system is thus relatively shallow and wide, and less overhead is introduced.

3. Polymorphism. Whenever appropriate, methods are declared virtual, with the goal to promote polymorphism. This was designed for inheritance by users, though the implementation of MPI++ itself did not need this.

4. Type signatures and overloading. C++ features like operator overloading and default arguments are used to provide simpler and more intuitive functions. For example, we have overloaded some operators to provide the same functionality as some of the group operations in the C binding of MPI. For example, the overloaded `-` operator is the same as the `MPIX_Group::Difference` member function. Others included the `|` and `&` operators for `MPI_Group::Union` and `MPIX_Group::Intersection` respectively, the `<<` and `>>` operators which shift a specified number of group members off the front or end of the group (*e.g.*, `group<<3` shifts the first three members off the front of the group), and the assignment versions of these operations (*i.e.*, `+=`, `-=`, `<<=`). As another example, the `tag` argument in communication calls is usually an optional argument that is set to `0` by default. The following code fragment is from the example in Section 2.1 and shows `MPIX_Comm::Send` and `MPIX_Comm:Recv` calls, which omit the `tag` argument.

```
if (rank == 0) {
  strcpy(message,"Hello, there");
  MPI_COMM_WORLD.Send(message,strlen(message),MPI_CHAR,size-1);
}
else if (rank == (size-1)){
  MPI_COMM_WORLD.Recv(message,MAX_MESSAGE_SIZE,MPI_CHAR,0,&status);
  cout << "received" << message << endl;
}
```

This is useful because many parallel algorithms emphasize source selectivity of messages.

2.3 Implementation Issues

With the above class hierarchy set up, it is often straightforward to implement the C++ methods using MPI C functions. Moreover, in as much as methods are made in-line to avoid significant overhead, the high performance achieved by MPI C interface will not be affected greatly.

As a consequence of this implementation strategy, to use MPI++, you have to have an existing MPI library running on your system, as well as a modern sequential C++ compiler. A model implementation of MPI, MPICH, is available in the public domain (see Section 1.3 for more information), and several commercial versions are appearing at this time.

If you are using MPI++ with MPICH, the MPI++ library and include files should be installed in the same locations as their MPI equivalents. Thus, the compile commands should look something like the following:

```
CC  -I/usr/local/mpi/include -c prog.C
CC  -o prog  prog.o -L/usr/local/mpi/lib -lmpi++ -lmpi
```

The above commands assume that the include files are in `/usr/local/mpi/include` and that the MPI and MPI++ libraries are in `/usr/local/mpi/lib`.

At the time of this writing, only the Gnu C++ compiler g++, the SunSPARCworks CC, and IBM xlC have been tested with this code, and MPI++ has been tested to work on Sun workstations and SP2. Since MPI++ uses standard C++, it is expected that it will work on other platforms too.

3 Examples

In this section, we will concentrate on the polygon overlay problem to illustrate more features of MPI++. In particular, we shall show how encapsulation and inheritance help to manage complexity. The solutions given below are here to illustrate the use of MPI++.

3.1 Solution #1

This solution is a usual master-slave implementation [15]. The master process reads in the polygons from the first map and distributes them evenly to all the slave processes. Then polygons from the second map are read in one at a time and are broadcasted to each of the slaves from the master. The intersection of each with each polygon in the first map is computed in parallel, and the results are sent back to the master process everytime and written to a file.

First we give a sketch of a possible "C-mindset" version of the solution using MPI++.

```
#include "mpi++.h"
#include <iostream.h>
const int COORD = 4;
main(int argc, char **argv)
{
    MPI_COMM_WORLD.Init(&argc,&argv);
    int rank,size;
    MPI_COMM_WORLD.Rank(&rank);
    MPI_COMM_WORLD.Size(&size);
    // The master process
    if (rank == 0) {
        // Open files, reads in the number of polygons in the first map
        // allocate memory, etc.
        .
        .
        .
        // Broadcast the number of polygon of the first map
        MPI_COMM_WORLD.Bcast( &count, 1, MPI_INT, 0 );
        // Declarations for nonblocking send
        MPI_Request_collection  request(size-1);
        MPIX_Status_collection  status_coll(size-1);
        // read in and send to each processor its share
        // of polygons (of size its_share)
        .
        .
        .
        MPI_COMM_WORLD.Isend (vect, its_share, MPI_INT, i, &request[i-1]);
        // Close file of the first map and wait for the
        // completion of all sends
        .
        .
        .
        request.Waitall( &status_coll);
        // Broadcast one polygon of the second map each time to every
        // slave process till the end of the second map
        .
        .
        .
        MPI_COMM_WORLD.Bcast( buffer, COORD, MPI_INT, 0 );
        //Gather results from each slave and write to file, the number of
        // intersection is found through status.Get_count
        while ( number_done < size-1 ) {
```

```
            MPI_COMM_WORLD.Recv( vect, count, MPI_INT,
                                 MPI_ANY_SOURCE, &status);
            status.Get_count( MPI_INT, &number_of_intersection) ;
            .
            .
            .
        }
        //Close files and tell slave processes that it is done
        buffer[0] = -1;
        MPI_COMM_WORLD.Bcast( buffer, COORD, MPI_INT, 0);
    }
    else { // slave process
        // Wait for the number of polygons of the first map and compute
        // its share of polygons, allocate memory and receive polygons
        MPI_COMM_WORLD.Bcast( &count, 1, MPI_INT, 0 );
        .
        .
        .
        MPI_COMM_WORLD.Recv(vect, my_share*COORD, MPI_INT, 0, &status);
        // While not end, receive one polygon of the second map and
        // do comparison, send results back to master process
        do {
            inter_count = 0;
            MPI_COMM_WORLD.Bcast( buffer, COORD, MPI_INT, 0 );
            .
            .
            .
            MPI_COMM_WORLD.Send( result_vect, inter_count*COORD,
                                 MPI_Int, 0);
        } while(buffer[0]>0);
    }
    MPI_COMM_WORLD.Finalize();
    return(0);
}
```

As you can see, as problems get more complicated, the "big" if-else structure is progressively harder to manage. More importantly, supposing there were more process types other than master and slave, the program could become hard to read.

The next version uses classes to manage processes. Conceptually, all the processes are similar, each has to go through a few phases, such as a set up phase, a work phase, and a finish phase. Their differences are encapsulated in the individual methods, and "virtual constructors"[5] will create the appropriate instance for each process. A sample code might look like the following.

```
// main program-------------------------------------//
#include "polygon.h"
main (int argc, char** argv)
{
    MPI_COMM_WORLD.Init(&argc, &argv);
    int myrank;
    MPI_COMM_WORLD.Rank( &myrank );
```

```
    General_process process( myrank, argc, argv );
    process.SetUp();
    process.SendData_work();
    process.SendRecvData_work();
    process.Finish();
    MPI_COMM_WORLD.Finalize();
    return( 0 );
}
```

The header file for this example is:

```
// the head file polygon.h ----------------------------------//
#ifndef  __polyV__H
#define  __polyV__H

#include <iostream.h>
#include "mpi++.h"
#include <stdio.h>
class Process {
public:
    // Constructors
    Process() { MPI_COMM_WORLD.Size( &size ); }
    // Destructor
    ~Process() { }
    virtual void SetUp( ) { }
    virtual void SendData_work( ) { }
    virtual void SendRecvData_work( ) { }
    virtual void Finish( ) { }
  protected:
    int size, rank, total_count, share_count, extra_count, *vect;
    MPI_Status status;
    int argc;
    char** argv;
};
class Master_process : public Process {
  public:
    Master_process(int r, int c, char** v) { rank = r; argc = c; argv = v;}
    ~Master_process ( ) { }
    void SetUp( ); // Do appropriate set up for master process
    void SendData_work( );   // Do appropriate work for master process
    void SendRecvData_work( ); // Do appropriate work for master process
    void Finish( );
  private:
    int length;
    FILE *fp1, *fp2;
};
class Slave_process : public Process {
  public:
```

```
    Slave_process(int r, int c, char** v) { rank = r; argc=c; argv=v;}
    ~Slave_process () { }
    void SetUp( ); // Do appropriate set up for slave process
    void SendRecvData_work( );// Do appropriate work for slave process
    // SetUp() and Finish() are not over written for slave process
  private:
    int my_count, *result_vect;
};
class General_process : public Process {
  public:
    General_process( int r, int c, char** v ) {
        if ( r == 0 )
            repProcess = new Master_process( r,c,v );
        else
            repProcess = new Slave_process( r,c,v );
    }
    // Destructor
    ~General_process() { delete repProcess; }
    void SetUp( ) { repProcess->SetUp(); }
    void SendData_work( ) {repProcess->SendData_work();}
    void SendRecvData_work( ) {repProcess->SendRecvData_work();}
    void Finish( ) {repProcess->Finish();}
  private:
    Process *repProcess;
};
inline void Master_process :: SetUp( )
⋮
```

For a simple problem like the polygon overlay application, this may not seem to gain us much. But certainly this is more manageable, as we can work on each method at a time, which is important in dealing with large, complex problems. Moreover, the extensibility and reusability introduced by this approach could prove valuable in building libraries or in an environment where applications always have certain fixed patterns [9].

3.2 Solution #2

The previous solution uses the default communicator `MPI_COMM_WORLD`. `MPI_COMM_WORLD` is OK to use for simple codes but often you may need more communicators. For instance, sometimes a process needs to communicate within a local group as well as globally. Also, in writing libraries, one usually needs to duplicate the input communicator and only works with the duplicate for "safety."

To illustrate these ideas as well as the use of inheritance with MPI++, we provide a pipeline controlled-parallel implementation to the polygon overlay problem. There are two controllers that will distribute data, the first one reads

in the polygons from the first map and distributes them to all the pipeline workers, and the second one is responsible for sending polygons from the second map to the first pipeline worker from there the polygons stream through the pipeline. The results are stored locally till a buffer is filled and then sent back to the first controller and written to a file. In our implementation, we first duplicate the input communicator `MPI_COMM_WORLD` to get a new communicator `NewWorld` where all the processes continue to belong. Then it is split into a controller communicator (it has all the controller processes) and a worker communicator (it contains all the pipeline worker processes) where controllers and workers belong, respectively. There is also a inter-communicator `inter` which can be used to communicate between controllers and workers.

Before we get into the details, first let us look at the ways to create new communicators by an example:

```
#include <mpi++.h>
main (int argc, char **argv)
{
  char       message[MAX_MESSAGE_SIZE];
  int        rank;
  MPI_Status status;
  MPI_Group  group;

  MPIX_COMM_WORLD.Init (&argc, &argv); // Collective
  MPIX_COMM_WORLD.Rank (&rank);        // Local
  MPIX_COMM_WORLD.Group(&group);       // Local

  // The next two lines of code are equivalent: world1 and world2 are
  // dups of MPI_COMM_WORLD
  //   MPI_Comm_dup(MPI_COMM_WORLD, &comm_out);
  MPIX_Comm_intra world1(MPIX_COMM_WORLD);
  MPIX_Comm_intra world2;  MPIX_COMM_WORLD.Dup(&world2);

  // The next two lines of code are equivalent:  split_world1 and
  // split_world2 will each contain 1 process.
  //   MPI_Comm_split(MPI_COMM_WORLD, rank, 0, &comm_out);
  MPIX_Comm_intra split_world1(MPIX_COMM_WORLD, rank, 0);
  MPIX_Comm_intra split_world2;
  MPIX_COMM_WORLD.Split(rank,0,&split_world2);

  // The next two lines of code are equivalent: group_world1 and
  // group_world2 will be copies of MPI_COMM_WORLD
  //   MPI_Comm_create(MPI_COMM_WORLD, group, &comm_out);
  MPIX_Comm_intra group_world1(MPIX_COMM_WORLD, group);
  MPIX_Comm_intra group_world2;
  MPIX_COMM_WORLD.Create(group,&group_world2);
  MPIX_COMM_WORLD.Finalize();
  return (0);
```

}

Here is a sketch of part of our implementation of the pipeline version. For the sake of brevity, only the main ideas are presented here.

```
//==================== world.h ====================
#ifndef  __WORLD__H
#define  __WORLD__H

#include "mpi++.h"
#include <iostream.h>
class NewWorld {
public:
    // Constructors
    NewWorld(  MPIX_Comm_intra& communicate ) {
        communicate.Dup ( &world );
    }
    // Destructor
    ~NewWorld(void) { world.Free(); }
    // Customized accessors
    MPIX_Comm_intra& World( )  { return world; }
    int Rank( ) { int rank;world.Rank( &rank );return rank;}
    int Size( ) {int size;world.Size( &size );return size;}
private:
    MPIX_Comm_intra world;  // Contains all processes.
};
class Controller : public NewWorld {
public:
    // Constructors
    Controller ( MPIX_Comm_intra& comm, int color) : NewWorld(comm) {
        World().Split( color,  Rank(), &local);
        int remote_leader = 2;
        local.Intercomm_create( 0, World(), remote_leader, 0, &inter );
    }
    // Accessors
    MPIX_Comm_intra& Local( )  { return local; }
    MPIX_Comm_inter& Inter( ) { return inter; }
    int Local_Rank() { int rank; Local().Rank( &rank );return rank;}
private:
    MPIX_Comm_intra local;   // Contains all controllers.
    MPIX_Comm_inter inter;  // Contains inter-communicator communication
                            //    with workers.
};
class Worker : public NewWorld {
public:
    // Constructors
    Worker ( MPIX_Comm_intra& comm, int color) : NewWorld( comm ){
```

```
        World().Split(color, Rank(), &local );
        int remote_leader = 0;
        local.Intercomm_create(0, World(), remote_leader, 0 , &inter);
    }
    // Accessors
    MPIX_Comm_intra& Local( )   { return local; }
    MPIX_Comm_inter& Inter( ) { return inter; }
    int Local_Rank() {int rank; Local().Rank( &rank );return rank;}
  private:
    MPIX_Comm_intra local;   // Contains all workers.
    MPIX_Comm_inter inter;   // Contains inter-communicator for
                             // worker-controller communication
};
#endif
```

Finally, in the main program, a process can use the appropriate communicator for communication. For instance, the controller responsible for distributing polygons of the first map only need send the number of polygons of the first map to all the workers, so a broadcast to every process in the world is not necessary. Instead, it can use the inter-communicator, sending a message to one of the workers, then that worker can broadcast the message to all workers using the communicator local to the workers[2]. In fact, since each process belongs to a local communicator and an inter-communicator, it knows how to communicate with others, so each can be developed, compiled, and executed as individual program. Details can be found from the source code (see Section 1.3).

This approach of arranging processes into groups associated with different communicators demonstrates one way of taking advantage of MPI++'s class hierarchy. For more complex problems, it will prove to be even more valuable. Moreover, this approach provides clean and readable parallel codes. The user can customize the class interface with great flexibility. For instance, in the sample above, we defined an accessor method in the base class `NewWorld` that gives the global process rank:

```
int NewWorld::Rank();
```

And we also defined an accessor method in the derived classes that gives the process rank within its local communicator:

```
int controller::Local_rank();
int worker::Local_rank();
```

Though we can use the accessor methods provided by MPI++ each time to achieve the same effect, these user-defined methods are much more handy to use, and usually more meaningful in a particular application. The user can also provide his own member functions appropriate to his application. For example, in the above `worker` subclass, member functions could be created for receiving

[2]In MPI-2, intercommunicators will support cross-group broadcasts.

data from the previous worker in the pipeline and for sending data to the next worker in the pipeline. Furthermore, member functions could also be defined for returning results to the controller process that is collecting data.

4 Performance Tests

In this section, we present some performance data for the polygon overlay application. Two programs, one based on the master-slave implementation and the other based on the pipeline implementation described in the previous section, were compiled using mpCC with the -g option, and run on IBM SP-2. The model implementation of MPI, MPICH [7], was used with MPI++.

Two maps were randomly generated on 4000×4000 grids, with one containing 203,483 polygons and the other containing 203,456 polygons. For the purpose of illustration, these maps were not sorted, and the two programs simply compared each pair of polygons from the two maps to find all the overlaps. For comparison, this same strategy was also used in the sequential program.

Table 2 lists some sample timings. The data in the table do not include any I/O time. In the master-slave implementation, one process (master) was solely for distributing data, so the number of processes actually computing was one less than the total number of processes. In the pipeline implementation, two processes were used only for data distribution, so the number of processes actually computing was two less than the total number of processes.

Table 2 shows that for small number of processors, the master-slave implementation used less processing time than the pipeline implementation, while for larger number of processors, the pipeline implementation used less processing time than the master-slave implementation. This is what one would expect (and is typical of the polyalgorithmic nature of parallel systems). When the number of processors used is small, the overall overhead of communcation is low, the master-slave implementation which has one more processor for computing runs faster. But as more processors are used, the overhead of communication becomes significant, thus the pipeline implementation which has less communication turns out to be better. A buffer of size 1000 polygons was used locally for each pipeline worker process in our sample program.

5 Summary, Conclusions, and Future Work

In the foregoing, we have introduced MPI++ at several levels:

- background on MPI++ (origins of this work, basis in the MPI system),
- parallel programming models supported (as inherited from MPI),
- basic capabilities (and some plausible uses),

Number of Processors used	Number of Processors Actually computing	Algorithm Used	Time (Seconds)
1	1	Sequential	22654.5
5	4	Master-Slave	4728.2
5	3	Pipeline	6081.89
6	5	Master-Slave	3844.6
6	4	Pipeline	4568.75
7	6	Master-Slave	3274.24
7	5	Pipeline	3663.39
8	7	Master-Slave	2884.39
8	6	Pipeline	3071.97
15	14	Master-Slave	1693.04
15	13	Pipeline	1591.29
25	24	Master-Slave	1189.66
25	23	Pipeline	840.679
35	34	Master-Slave	995.389
35	33	Pipeline	616.129
45	44	Master-Slave	1051.03
45	43	Pipeline	467.873

Table 2: Polygon overlay speeds

- design restrictions we imposed on ourselves,
- the syntax and semantics that resulted,
- MPI++ implementations of the polygon overlay application, and
- some performance results for the polygon overlay application on the IBM SP-2.

Unlike other systems described in this book, MPI++ provides a full parallel programming model without the benefit of compiler support. It is exceptionally portable, as it can function wherever MPI runs. As MPI implementations are rapidly improving, the performance potential of MPI++ applications appears extremely positive.

In this section, we consider some potential criticisms of MPI++, as it now stands, and comment on the efforts on-going in MPI-2, since they impact MPI++ in two ways: first, new functionality in MPI would have to be represented in a future release of MPI++, and, second, a C++ language binding is being considered for MPI-2. This work is directly related to that language-binding effort; one outcome would be that part or all of MPI++ became standardized in the MPI-2 C++ language binding. In such a case, this project and its related effort, MPICH, would effectively merge. Alternatively, a quite different C++ language binding might emerge which could lead to us discontinuing

MPI++, or providing it as a significant alternative to the MPI language binding, depending on the functionality that is chosen for this binding. Finally, the MPI Forum might choose to drop its C++ language binding effort, leaving the C binding as the base software for C++ development.

5.1 Critical Look at Current MPI++

5.1.1 Current Design

The current design of MPI++ made rather 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 quite C-like in nature. However, in several cases, the class structure of C++ was exploited successfully.

What is absent here, largely because MPI objects are already reference-counted, is elaborate discussions of nested classes, inheritance, and the handle/body (or envelope/letter) idioms discussed by Coplien [5]. The MPI++ objects that implement opaque MPI objects are relatively simple because the objects from the C language binding already are 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 are inevitable. For instance, it is not possible to layer certain calls involving arrays of request objects on top of the MPI equivalent call[3]; specifically one can not pass an array of C++ requests easily to the C routine, because the objects would differ in content and format. Hence, either the use of certain classes of MPI functions will have to be replaced by loops over simpler versions of these functions at the C++ level, or expensive object reorganizations will be needed on entry and exit from such routines. In either event, performance is bound to be diminished, and one risks a change in semantics from the MPI C binding for such functions if they are reimplemented in a specific way by MPI+, using more primitive MPI calls.

The solution to this problem is to consider a ground-up implementation of MPI in C++, basing the system on a small, abstract device, as currently done in the C version. However, if the entire system were C++, rather than just the uppermost layers, considerable improvements would potentially result, 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.

5.1.2 Generalized Parallel Application Architecture

Finally, the current MPI++ system does not alone support the full parallel application architecture that we are convinced is needed for practical applications (shown in Figure Figure 7). This application architecture is certainly attainable

[3]Compare `MPI_Wait` and `MPI_Waitall`, to be specific.

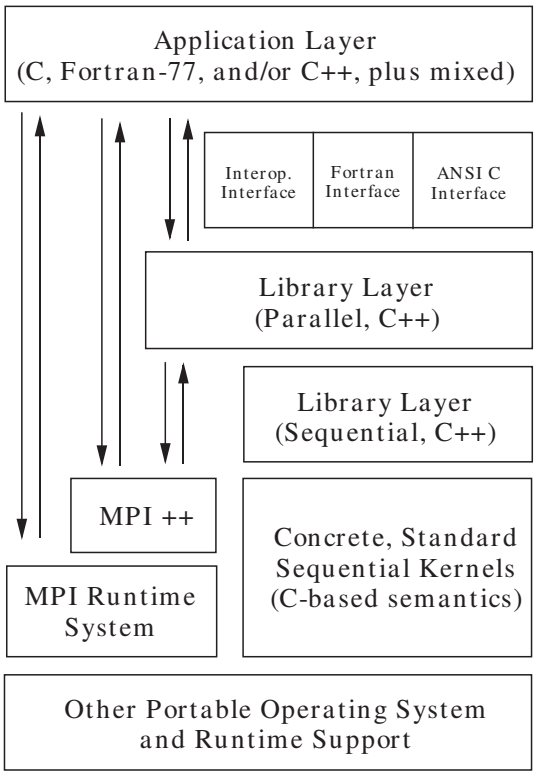

Figure 7: For the future, this is the canonical parallel application architecture we wish to support, showing the role of MPI++ within that hierarchy.

given current technology and our understanding of how to build libraries. Significant infrastructure must, however, be created in order to provide the needed critical mass to enable real applications in this environment.

5.1.3 Application Experience

As a further limitation of the current system, relatively trivial amounts of code have been written in MPI++ as compared to MPI programs written in C and Fortran. Hence, we lack the perspective 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 into MPI++. (For instance, we have not provided for multiple inheritance based on our classes, though we follow Coplien's *Inheritance Canonical Form* for simple inheritance [5].) We might

further 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 further impetus for change and improvement.

5.2 MPI-2

The MPI Forum reconvened in March of 1995 to begin discussions about extending the MPI Standard. One of the guiding principles for "MPI-2" meetings is not to change the MPI-1 Standard, but to extend it, except if grave errors in MPI should be demonstrated. The Forum is divided into the following subcommittees:

Dynamic dynamic process management

One-Sided one sided communication such as put/get and active messages

Collective intercommunicator and other extensions to collective operations

External extensions for debugger and profiler support

Bindings bindings for languages other than C and Fortran

Real-time extensions for real-time and embedded computing

Miscellaneous miscellaneous corrections and clarifications to the MPI-1 standard.

At the time this chapter was written, the bindings committee was considering Ada-95, Fortran 90, and C++ bindings as a part of MPI-2. The remainder of this section provides an overview of the MPI-2 language binding committee's discussion about C++. Further information about MPI-2 and about the language binding committee in particular including meeting minutes and mail reflectors is available through the WWW [1].

5.3 C++ Bindings and C++ Class Library

The C bindings provide one interface for writing MPI programs in C++. C++ MPI applications may be written by simply including the "mpi.h" header file and calling MPI functions as described by the C bindings. Although one can write C++ applications using the C bindings, a higher level C++ class libary is being considered for inclusion in the MPI-2 standard. The general consensus of the committee is that a class library is important for promoting the use of MPI among people coming from an object-oriented background and for broadening the scope for parallel programming in object-oriented languages based on MPI. It is also believed that a class library allows the programmer to program at a

higher level of abstraction thereby allowing the programmer to spend more time on problem-specific issues.

As we have already observed, the MPI specification is object-based with MPI handles such as requests, communicators, groups, and so on as the objects in the system. Early proposals have advocated that these handles become the C++ classes that make up the MPI C++ class library, similar to what we have already done. Constructors, destructors, accessors, and functions that operate on the MPI handles become methods of these classes. Concepts not explicitly present in the MPI standard are not likely to be included in an MPI-2 class library.

5.4 Should the MPI-2 Standard Contain a C++ Class Library?

The MPI Forum has not voted on whether or not a C++ class library will be included in the MPI-2 Standard. The following are some of the pros and cons brought up in committee discussions concerning the issue of including a C++ class library definition in the MPI-2 standard.

Pros of having an MPI C++ class library defined in MPI-2:

- A standardized class library would have a better chance of promoting the use of MPI among people coming from an object-oriented background and broadening the scope for parallel programming in object-oriented languages based on MPI.
- Lack of a standardized class library would not satisfy the object-oriented communities expectations and would leave them with three alternatives:
  - not to use MPI
  - invent their own higher level abstractions — each group inventing different abstractions
  - form a group to define a common, higher level binding.

  The MPI Forum is probably the best community to sponsor the third alternative, thus saving others from the negative impacts of the second alternative.
- If we don't provide a C++ class library as part of the standard, there will be many non-standard MPI C++ class libraries.

Cons of having an MPI C++ class library defined in MPI-2:

- A C++ class library for MPI is something more than bindings. Although a class library is an important effort, it is beyond the scope of simply language bindings for MPI. The general specification of MPI is (intentionally)

language- and implementation-independent. The bindings provide the interface between the MPI functionality and a real language that wants to use that functionality. As such, MPI bindings should only provide the interface between a C++ class library and MPI.

- A class library specification will necessarily involve a significant amount of implementation detail, unlike any other part of the MPI specification thus far.
- One of MPI's primary goals was to support the development of portable libraries. Standardizing a library that can be easily built on top of MPI is unnecessary and should be considered syntactic sugar.
- It is beneficial that there be many MPI C++ class libraries. A likely outcome would be many portable MPI C++ class libraries, each of which provides a different useful set of abstractions.

5.5 Conclusions

We hypothesized at the outset that a good class library in C++ based on MPI could provide an acceptable to excellent parallel C++ environment. We have motivated this throughout the foregoing discussion. What remains to be seen, and studied in detail, are the unique features that a compile-based environment can provide that C++, as extensible as it is by users, cannot provide. We look forward to such a comprehensive study, as we believe this will help drive further research into the most fruitful directions in 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.

For now, we are content to continue to work with our library-based approach, working to improve its functionality, while retaining its good performance-portability. We furthermore plan to investigate a fully C++ implementation of MPI as a revolutionary step in this research area.

References

[1] (Anonymous). MPI-2 World Wide Web Page. http://www.erc.msstate.edu/mpi/.

[2] Purushotham Bangalore, Nathan Doss, and Anthony Skjellum. MPI++ : Issues and Features. In *Proceedings of OONSKI '94*, January 1994.

[3] Ralph Butler and Ewing Lusk. A user's guide to the P4 parallel programming system. ANL 92/17, Argonne National Laboratory, Argonne, Illinois, October 1992.

[4] R. Calkin, Hoppe H.-C. Hempel, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, 1994.

[5] James O. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley Publishing Company, 1992.

[6] O. Coulaud and E. Dillon. PARA++ C++ bindings for message passing libraries: User guide. Technical Report INRIA-n XXXX, National Research Institute in Software Engineering and Automatism, France, April 1995.

[7] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A Model Implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993. in preparation.

[8] Robert D. Falgout, Anthony Skjellum, Steven G. Smith, and Charles H. Still. The multicomputer toolbox approach to concurrent BLAS and LACS. In J. Saltz, editor, *Proc. Scalable High Performance Computing Conf. (SHPCC)*, pages 121–128. IEEE Press, Los Alamitos, CA, 1992. Also available as LLNL Technical Report UCRL-JC-109775.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Publishing Company, 1995. ISBN 0-201-63361-2.

[10] A. Geist, A. Beguelin, J. Dongarra, J. Weicheng, R. Manchek, and V. Sunderam. PVM 3 user's' guide and reference manual. ORNL/TM 12187, Oak Ridge National Laboratories, Oak Ridge, TN, 1993.

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

[12] William D. Gropp and Barry Smith. Chameleon Parallel Programming Tools User's Manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[13] Shang-Hsien Hsieh and Elisa D. Sotelino. PPI++: An object-oriented parallel portability interface in C++. Technical report, School of Civil Engineering, Purdue University, West Lafayette, Indiana, 1994.

[14] Ken Kennedy, editor. *Proc. of First CRPC Workshop on Standards for Message Passing in a Distributed Memory Environment.* CRPC, April 29-30 1992.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[16] Message Passing Interface Forum. *MPI2: Draft Document for the MPI2 Standard*, October 1995. Available from ftp://ftp.super.org/pub/mpi/mpi2/Editor/mpi2-report.ps.

[17] Anthony Skjellum and Chuck Baldwin. *The Multicomputer Toolbox:* Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[18] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The design and evolution of Zipcode. *Parallel Computing*, 20:565–596, 1994.

[19] André Weinand, Erich Gamma, and Rudolf Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2), 1989.