MPI++: Issues and Features*

Purushotham V. Bangalore Nathan E. Doss Anthony Skjellum†

Department of Computer Science,
Integrated Concurrent & Distributed Computation Research Laboratory
and
NSF Engineering Research Center for Computational Field Simulation
Mississippi State University
Mississippi State, MS 39762

Abstract

The draft of the MPI (Message-Passing Interface) standard was released at Supercomputing '93, November 1993. The final MPI document is expected to be released in mid-April of 1994. Language bindings for C and FORTRAN were included in the draft; however, a language binding for C++ was not included. MPI provides support for datatypes and topologies that is needed for developing numerical libraries, however the interfaces described in the MPI draft document for these two features have several disadvantages.

In this paper we describe and offer examples of a C++ interface to MPI that can be used to build scalable, object-oriented numerical libraries in a language that directly supports object-oriented programming. We also introduce several ideas from the *Zipcode* message-passing library related to datatypes and topologies, then compare and contrast these to the MPI approach. The best ideas from both approaches are combined to make a better interface to datatypes and topologies.

*Work supported by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University. Revision 1.00. March 14, 1994.

†Author to whom correspondence should be addressed: *tony@cs.msstate.edu*. (601)325-8435.

1 Introduction

There are many message-passing libraries such as PVM [1], P4 [4], *Zipcode* [13], PICL [9], and PARMACS [5] that can be used (with varying degrees of difficulty) to write parallel numerical libraries. Most of these provide both C and FORTRAN language interfaces; few if any provide interfaces to an object-oriented language. MPI (Message-Passing Interface) is a standard message-passing notation that was developed over the last one and a half years with input from industry, national laboratories, and universities [8]. The draft of the MPI standard was released during Supercomputing '93 (November, 1993), with the final draft expected in mid-April of 1994. MPI provides many of the features needed to build portable, efficient, scalable, and heterogeneous message-passing code. These features include point-to-point and collective communication, support for datatypes (scatter/gather specifications), virtual topologies, process-group and communication-context management, and language bindings for both FORTRAN 77 and C.

MPI does not, however, provide an interface to an object-oriented language though the draft does suggest that experiments with C++ language bindings should be undertaken [8, page 9]. A C++ interface would allow programmers to write message-passing code for object-oriented numerical libraries in a language that directly supports an object-oriented programming style. One purpose of this paper is therefore to present our work on designing and experimenting with a C++ language binding for MPI.

Support for datatypes and topologies is important for developing numerical libraries, however, the interfaces described in the MPI draft document for these features are somewhat difficult as well as awkward to use. Another goal of our work is therefore to introduce ideas on datatypes and topologies from *Zipcode* into MPI. *Zipcode* invoices are not as expressive as MPI datatypes, but in most cases are simpler to use. *Zipcode* topologies are simpler to use but are not as general as MPI topologies. By combining ideas from *Zipcode* and MPI, we are able to provide an interface to datatypes and topologies that combines the benefits of both approaches.

This paper assumes the reader has some knowledge of MPI, but provides most of the background necessary to understand the majority of the discussion on the C++ interface, datatypes, and topologies. We first discuss our C++ interface to MPI and show some simple examples of it. We look at the MPI and *Zipcode* implementations of datatypes and topologies, compare and contrast the two approaches, and present our approach for combining ideas from both. Our conclusions include suggested future work in this area, and potential improvements to what we have presented here.

2 A C++ Interface to MPI

The main goal of our C++ interface to MPI is to provide an extensible, portable, efficient, heterogeneous, and object-oriented message-passing library that can be used to build scalable, object-oriented numerical libraries. We already have a collection of object-oriented numerical libraries in the *Multicomputer Toolbox* [7, 10, 12] that were originally written in C using *Zipcode*. Even though the *Toolbox* libraries were written using an object-oriented design, they lack several of the niceties that result from writing in a language that directly supports an object-oriented style of programming. Our first step in converting the *Toolbox* libraries to C++ is "MPI++."

The MPI specification was developed using an object-oriented approach. It is thus relatively easy to determine the major classes and their interactions with each other. The main class in MPI is "MPI_Comm" (a communicator object). Constructors and destructors for communicators are provided along with operations for:

- point-to-point messages passing [8, section 3],
- collective communication [8, section 4],
- topology manipulation [8, section 6],
- attribute caching [8, section 5].

MPI communicators provide a safe scope for message passing. They consist of a "static process group" (a list of process-group members that may participate in a communication operation on a communicator) and a "context" of communication (a system-managed tag that is used to ensure that messages in one communicator do not interfere with messages in another). MPI provides an initial communicator (MPI_COMM_WORLD) that contains all processes in the initial runtime environment.

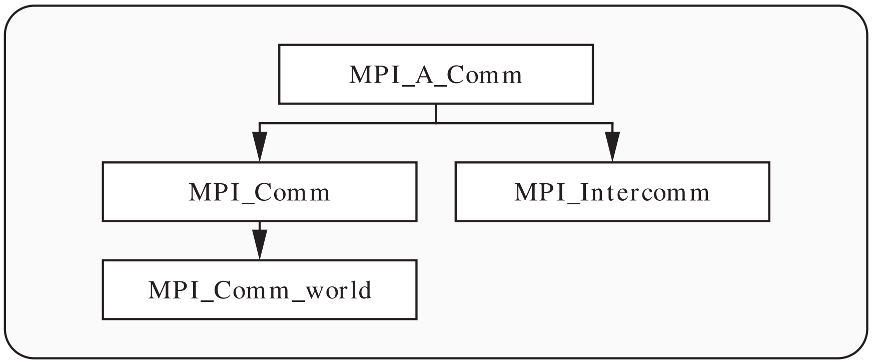

Figure 1: Hierarchy of communicator-based classes.

Figure 1 shows our interpretation of the hierarchy of communicator classes in MPI. Each type of communicator class only has those functions that are directly usable by the class. The `MPI_A_Comm` class is an abstract class from which the `MPI_Comm` (intra-communicators) and `MPI_Intercomm` (inter-communicators) classes are derived. `MPI_A_Comm` contains point-to-point and accessor functions that are common to both intra- and inter-communicators. The `MPI_Comm` class contains collective communication and topology functions as well as various other functions that do not apply to inter-communicators. `MPI_Comm_world` is a special class derived from `MPI_Comm` that additionally contains functions for manipulating the environment (*i.e.*, initialization and termination). `MPI_COMM_WORLD` is the only instance of this class (in the current MPI specification, but we expect that future drafts will permit multiple "worlds").

Two other important classes are `MPI_Group` and `MPI_Datatype`. Groups are manipulated with operations such as group addition and subtraction, and various set-like[1] operations. Datatypes are used to specify the type and location of data to be sent in messages. MPI provides several constructors for building different kinds of datatypes such as contiguous and vector types. Datatypes may also be built recursively (analogous to ELROS, see [2, 3]).

An initial C and FORTRAN implementation of MPI is essentially complete [6]. Our current C++ interface is implemented by providing wrappers around the C interface. The C and FORTRAN interfaces described in the draft use the function names to distinguish the object upon which the function works. Since this is not needed in C++ we have shortened the names of most functions. Most functions retain the same arguments as specified in the C binding.

Although we have not fully explored the use of many object-oriented features provided by C++, we note that features such as operator overloading and default arguments can be used to provide simpler and/or more intuitive functions. For example, when manipulating groups,

```
// Current C++ interface
group1.Union(group2, group_out);
```

could be written as:

```
group_out = group1 | group2;
```

These types of convenience features will be added incrementally to the MPI++ interface.

Another important item to note is that inheritance provides a means for the user to add new functionality to existing MPI++ classes with relative ease. The discussion of virtual topologies in MPI++ shows how inheritance can be used to create new topologies with a minimal amount of work (see section 4).

We close this section with a motivating example; figure 2 shows a simple example taken from the MPI draft document [8, section 5]. Figure 3 shows this same example when written in MPI++. The C++ version differs from the C version in several ways. For instance, a tag is not specified for the send and receive operations in the C++ version. The tag is an optional argument with a default value of 0. In both examples, `MPI_COMM_WORLD` is supplied as the initial communicator.

3 Datatypes

MPI datatypes and *Zipcode* invoices both provide descriptions of data to be sent in a message-passing operation. Each communication operation uses this "what" information to gather data to be sent or scatter

[1] Groups are sets plus a rank property for set members.

```
main(int argc, char **argv)
{
  int me, size;
  int TAG = 0;
  /* ... */
  MPI_Init (&argc, argv);

  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  if ((me % 2) == 0) {
     if ((me + 1) < size)
        MPI_Send (buf, count, type, me+1,TAG, MPI_COMM_WORLD);
  }
  else
     MPI_Recv (buf, count, type, me-1, TAG, MPI_COMM_WORLD);

  MPI_Finalize();
}
```

Figure 2: A C example from the MPI draft

```
main(int argc, char **argv)
{
  int me, size;
  // ...
  MPI_COMM_WORLD.Init(argc, argv);
  MPI_COMM_WORLD.Rank(me);
  MPI_COMM_WORLD.Size(size);

  if ((me % 2) == 0) {
     if ((me + 1) < size)
        MPI_COMM_WORLD.Send(buf,count,type,me+1);
  }
  else
     MPI_COMM_WORLD.Recv(buf,count,type,me-1);

  MPI_COMM_WORLD.Finalize();
}
```

Figure 3: An MPI++ example corresponding to Figure 2.

data upon receipt of a message. This frees the programmer from specifying "how" a message is packed. In MPI, datatypes are the only means available for specifying information to be sent and received in messages.

Several types of operations for creating and using invoices and datatypes are supported as follows:

- Constructors and destructors,
- Explicit packing and unpacking,
- Pack-sends and unpack-receives,
- Collective operations.

We illustrate these operations in the examples that follow in this section.

3.1 *Zipcode* Invoice Example

Zipcode invoices are created using a printf-like syntax. The "zip_new_invoice" call in Figure 4 is the constructor for invoices. Format strings are used to describe the type of data that comprises an invoice. They consist of a list of "formats" each beginning with a percent (%) sign. The last element of a format is a single letter that indicates the basic type of the elements to be included (*i.e.*, "i" for integer, "c" for character, ...). Between the percent sign and the type, optional arguments can be supplied to indicate the number and stride of the items. An address must be supplied for each format after the format string. The example given in Figure 4 creates an invoice that has the first ten integers from an array *a* and every other integer of the first twenty from an array *b*. Once an invoice has been constructed, it can be used in simple send and receive communication operations as well as collective operations such as "combine" (a reduce-all operation in MPI notation). For a full description of *Zipcode* invoices, see [13, 14].

```
zip_new_invoice(&inv,"%10i %10.2i",a,b);

g1_pack_send (mailer, inv, dest);
g1_pack_recv (mailer, source, inv);
g1_pack_combine (mailer, inv, method);
```

Figure 4: *Zipcode* invoice example

3.2 MPI Datatype Example

MPI provides functions for building contiguous, vector, indexed, and struct datatypes. In general, new MPI datatypes are built recursively from previously built datatypes. `MPI_Type_struct` may be used to concatenate two or more datatypes together. Figure 5 shows how to construct a datatype that contains the same information as the invoice created in Figure 4. The `MPI_Type_vector` call creates a datatype with a count of ten and a stride of two. The block, displacement, and type arrays passed to MPI_Type_struct contain:

block	displ	type
10	&a	MPI_INT
1	&b	vector

The MPI_Type_struct call uses these arguments to create the appropriate datatype. Once a datatype has been created, it must be committed (implementation-dependent optimization) by calling MPI_Commit before it can be used. After it is committed, the datatype may be used in send and receive communication operations as well as most of the collective operations. For more information on MPI datatypes consult the MPI draft [8, section 3].

```
MPI_Datatype type[2], vector;
int          block[2] = {10,1};
int          displ[2];

type[0] = MPI_INT;
MPI_Type_vector(10,1,2,MPI_INT,&vector);
type[1] = vector;
MPI_Address(a, displ);
MPI_Address(b, displ+1);
MPI_Type_struct(2, block, displ, type, &newtype);
MPI_Commit(newtype);

MPI_Send(start, count, newtype, dest, tag, comm);
MPI_Recv(start, count, newtype, source, tag, comm, status);
MPI_Allreduce(sendbuf, recvbuf, count, datatype, method, comm);
```

Figure 5: An MPI datatype example

3.3 Comparison of Invoices and Datatypes

The previous discussions have presented the basic syntax and ideas behind invoices and datatypes, but have not covered all their features and properties. The following list contains three of the most important differences between invoices and datatypes:

- MPI datatypes can be built recursively from other datatypes. Invoices are built by concatenating basic data types (int, char, float, etc.) together.
- Invoices may be bound to specific memory locations at the time they are created and any time thereafter. Datatypes are bound to specific memory locations when used in a communication operation.
- Datatypes are read-only, invoices may be re-sized.

Invoices are simpler to use than MPI datatypes. Datatypes have the following advantages:

- *Expressivity* MPI datatypes can describe data layouts that are either hard or impossible for invoices to describe. The two central features invoices lack that limits their expressivity is the inability to define new invoices recursively and the lack of a block size for strided types.

- *Orthogonality* All MPI communications use a "start location," a count of datatype elements, and the datatype. *Zipcode* provides one set of communication operations that takes invoices as arguments and another set that takes either a buffer pointer and a length (for collective operations) or a *Zipcode* letter (an encapsulated buffer object) for point-to-point operations.

3.4 Proposed MPI++ Datatype Interface

An improved MPI datatype interface should be simple to use, expressive, and orthogonal. In order to accomplish this, we use the invoice printf-like method to create datatypes. The syntax of the invoice format string is extended to include block lengths for strided items and to allow recursive building of invoices.

Datatypes in communication operations are defined by their relative position to the "start" argument. The MPI draft defines two variations on datatypes: absolute and relative datatypes [8, page 66]:

> We say that a datatype is **absolute** if all displacements within this datatype are valid (absolute) addresses: it is **relative** otherwise.

The draft also notes that absolute and relative datatypes have different properties and that some restrictions must be placed on their use. For example, all elements of a datatype must be either relative or absolute (not some combination of both), and the start argument must always be `MPI_Bottom` for absolute types. Other properties are listed on pages 65 and 66 of [8]. The interface to MPI datatypes does *not* make a distinction between absolute and relative types. By contrast, our interface to datatypes does make such a distinction by providing a separate class for each of these kinds of datatypes. The restrictions and properties of relative and absolute datatypes are built into the behavior of these classes.

The three new classes provided in our interface are `MPI_Relative`, `MPI_Absolute`, and `MPI_Buffer`. The first two classes are datatypes that provide the distinction between relative and absolute MPI datatypes. The `MPI_Buffer` class combines a datatype with a contiguous buffer and provides methods for packing (unpacking) `MPI_Relative` and `MPI_Absolute` types into (out of) the buffer. Figure 6 shows an example of our proposed MPI++ datatype interface.

The following example shows how to create a re-sizeable matrix datatype of size `rows` × `cols` recursively (assuming row-major storage). The size of the "matrix" datatype can be resized by changing the value of "rows" or "cols." The "#" sign in the format string indicates that the address of a variable is needed and "t" indicates that a previously defined `MPI_Relative` type is required.

```
MPI_Relative *row    =  new MPI_Relative("%#i",&rows);
```

```
MPI_Relative rtype("%10i");
MPI_Absolute atype("%10i %10.2i",a,b);

MPI_Buffer rbuf(rtype);
MPI_Buffer abuf(atype);

comm.Send(MPI_BOTTOM, 1, atype.type, dest);
comm.Recv(MPI_BOTTOM, 1, atype.type, src);

comm.Send(a, 1, rtype.type, dest);
comm.Recv(b, 1, rtype.type, src);

rbuf.pack(a, 1);
comm.Send(rbuf.start, rbuf.count, rbuf.type, dest);
comm.Recv(rbuf.start, rbuf.count, rbuf.type, src);

abuf.pack();
comm.Send(abuf.start, abuf.count, abuf.type, dest);
comm.Recv(abuf.start, abuf.count, abuf.type, src);
```

Figure 6: MPI++ datatypes example

```
MPI_Relative *matrix =  new MPI_Relative("%#t",&cols,row);
```

The next example shows how to create a datatype for the lower-triangular portion of a 5 x 5 square matrix.

```
MPI_Relative *lower  = new MPI_Relative("%5.5:(1-)i");
```

The "." is used to indicate that the stride of the type follows. ":" indicates that the next number in the format string is the block length. In this example, "(1-)" specifies an array of integers that begins at one with an increment of 1.

The `MPI_Absolute`, `MPI_Relative`, and `MPI_Buffer` classes can be layered on top of the existing MPI support for datatypes. However, in order to implement these efficiently, future versions of MPI should support access to system buffers and re-sizeable datatypes.

4 Virtual Topologies

Virtual topologies provide a machine-independent naming abstraction to describe communication operations in terms that are natural to an application. MPI and *Zipcode* both support the use of virtual topologies. Both systems have functions that map rank-in-group names to and from appropriate logical topology names. *Zipcode* and MPI topology examples are given in this section along with a comparison of the approaches in the two systems. Our approach to combining the best features of both concludes this section.

4.1 *Zipcode* Topology Example

Zipcode provides a specific set of topologies that can be used to write application code. The three most used classes are the grid classes – one-, two-, and three-dimensional cartesian topologies. In *Zipcode*, these are called the "g1", "g2", and "g3" grid classes. Figure 7 shows the creation and use of the g2 and g3 grid classes, respectively. The `grid_open` functions return a "mailer" that contains contexts and a static process group, similar to an MPI communicator. The grid send and receive operations take a number of arguments that are directly related to the dimension of the grid. For example, the `g2_send` call expects p and q arguments, that specify the location of the destination process in the logical 2D grid. The g1 and g3 communication operations expect arguments which specify the process location in 1D and 3D grids respectively.

In addition to contexts and a static process group, mailers also contain a hierarchy of other mailers. Each g3 mailer contains a g2 mailer for each plane of the grid to which the process belongs — a PQ, PR, and QR plane. All g2 mailers contain two g1 mailers. One of these g1 mailers consists of all members of the row of which the process is a part. The other g1 mailer contains all members of the process' column. Mailers in the hierarchy can be accessed by using macros provided by *Zipcode*. Once obtained, these mailers may be used in subsequent communication operations.

```
g2_mailer = g3_PQ_plane(g3_mailer);
g1_mailer = g2_col(g2_mailer);
```

Zipcode also provides shortcut notation that can be used directly in order to perform communication operations on lower-dimension mailers. Figure 7 illustrates the use of the `g2_row_combine` and `g3_QR_plane_combine` operations.

4.2 MPI Topology Example

MPI provides functions to create arbitrary graph topologies associated with a communicator. It also has functions that can be used to create n-dimensional cartesian topologies. Graph topologies are created by providing a set of nodes and edges. Figure 8 illustrates the creation and use of a 2D cartesian topology of dimension $P \times Q$. Cartesian topologies are created by specifying the number of dimensions along with arrays that contain information about the size and periodicity of each dimension.

Since all MPI communications expect a single integer rank-in-group for the source and/or destination of messages, topology-specific names must be translated to rank-in-group. MPI provides functions to perform this mapping (`MPI_Cart_rank`) as well as the inverse mapping (`MPI_Cart_coords`). The use of these two functions is demonstrated in Figure 8.

The `MPI_Comm_split` function in Figure 8 partitions the existing communicator into disjoint sub-communicators. The p argument to the `MPI_Comm_split` function specifies that the communicator should be partitioned so that processes in the same row are in the same communicator. The q argument causes

```
int      P, Q, R, p, q, r;

/* Create mailers */
mailer = g2_grid_open(&P,&Q,addressees);
mailer = g3_grid_open(&P,&Q,&R,addressees);

/* Send and Recv from member of 2D grid */
         g2_send (mailer, buffer, p, q);
letter = g2_recv (mailer, p, q);

/* Send and Recv from member of 3D grid */
         g3_send (mailer, buffer, p, q, r);
letter = g3_recv (mailer, p, q, r);

/* Perform a combine on row members */
g2_row_combine (mailer, buffer, method, size, nitems);
g3_QR_plane_combine (mailer, buffer, method, size, nitems);
```

Figure 7: *Zipcode* topology example

`MPI_Comm_split` to order the row processes according to their column number. Once the communicator is partitioned this way, the newly created "row" communicator is used to perform an independent reduce operation in each row of the topology (for more detail, see [11]).

4.3 Comparison of *Zipcode* and MPI Topologies

MPI provides functions that dynamically create new topologies while *Zipcode* only provides a specific set of pre-defined topologies. New topologies can be created relatively easily in MPI. In the current implementation of *Zipcode*, it is difficult to create new user-defined topologies.

Zipcode provides a natural way to specify process locations in terms of the topology whereas topology-specific names must be mapped to a rank-in-group in order to be used with MPI communication operations. This allows MPI to use the same communication operations (such as send and receive) for communicators with and without attached virtual topology information. In *Zipcode*, each topology class must have communication operations that accept the correct number of arguments for the particular topology.

Another important difference is that *Zipcode* mailers contain a hierarchy of mailers whereas MPI communicators only have information about one topology. (A hierarchy of MPI communicators can be layered on top of MPI to provide the same capability.)

5 Proposed MPI++ Topology Support

It is relatively easy to derive *Zipcode*-like grid classes from the `MPI_Comm` class. Figure 9 shows the class declaration for a simple 2D-grid class. The `MPIX_Grid2d` class is a communicator with topology information

```
int      P, Q, p, q;
int      coords[2], dims[2], period[2];

/* Create an MPI topology */
dims[0] = P; dims[1] = Q;
period[0] = period[1] = FALSE;
MPI_Make_cart(comm, 2, dims, period, TRUE, &comm_2d);

/* Send and Recv from member of 2D grid */
coords[0] = p; coords[1] = q;
MPI_Cart_rank(comm_2d, coords, &rank);
MPI_Send (start, count, datatype, rank, tag, comm);
MPI_Recv (start, count, datatype, rank, tag, comm);

/* Perform a reduce on row members */
MPI_Cart_coords(comm_2d, my_rank, 2, coords);
p = coords[0]; q = coords[1];
MPI_Comm_split (comm, p, q, &row_comm);
MPI_Allreduce(sbuf, rbuf, count, type, method, row_comm);
```

Figure 8: MPI topology example

and inherits from `MPI_Comm`. It also contains a row (column, resp.) communicator that consists of all processes in the same process row (column, resp.) of the grid. These communicators are used for communications that are local to the process row or column. The "Init" function initializes the grid to be of size $P \times Q$ and creates the row and column communicators. Point-to-point and collective functions that take rank-in-group arguments (*i.e.*, Send, Recv, Bcast, etc.) can be overloaded to accept grid-specific process names. For example, instead of sending a message to process 5, the message may be sent to a specific row, column coordinate such as $(5, 3)$. Accessor functions are also provided for accessing the dimensions of the grid and the position of the local process in the grid. Higher dimension grid classes can also be derived from the `MPI_Comm` class in a similar manner. Figure 10 illustrates the use of the `MPIX_Grid2d` and `MPIX_Grid3d` classes.

6 Future Work

A number of projects concern us immediately and for the near future. We will

- experiment further with the MPI++ interface and how we can better use the power of C++.
- recast the *Multicomputer Toolbox* libraries in C++ using MPI++ as the message-passing layer.
- write and distribute MPIX (MPI e"X"tension) libraries that will contain such things as:
  - the datatype classes described in this paper,

```
class MPIX_Grid2d: public MPI_Comm {
public:
  // Row and column communicators
  MPI_Comm Row, Column;

  // Constructors
  MPIX_Grid2d(void);

  // Destructor
  ~MPIX_Grid2d(void);

  // Free grid
  int Free(void);

  // Initialize grid
  int Init(MPI_Comm& comm_in, int P, int Q);

  // Duplicate grid
  int Dup(MPIX_Grid2d& grid_out);

  // Overloaded point-to-point operations
  int Send  (void *, int, MPI_Datatype, int, int, int);
  int Recv  (void *, int, MPI_Datatype, int, int, int, MPI_Status&);
  // etc.

  // Overloaded collective operations
  int Bcast  (void *, int, MPI_Datatype, int, int);
  int Reduce (void *, void *, int, MPI_Datatype, MPI_Op, int, int);
  // etc.

  // Grid accessors
  int P(void);
  int Q(void);
  int p(void);
  int q(void);

private:
  int P_, Q_;
  int p_, q_;
};
```

Figure 9: Class declaration for a 2D-grid class

```
MPIX_Grid2d Grid2d;
MPIX_Grid3d Grid3d;
int         size, rank, dims[2];

// ...

MPI_COMM_WORLD.Size (size);
MPI_COMM_WORLD.Make_dims (size, 2, dims);

// Initialize the grid
Grid2d.Init (MPI_COMM_WORLD, dims[0], dims[1]);

// Find my position in the grid
int p = Grid2d.p(); int q = Grid2d.q();

// Send a message to my right neighbor
Grid2d.Send (&data, 1, MPI_INT, p+1, q);

// Receive a message from my bottom neighbor
Grid2d.Recv (&data, 1, MPI_INT, p, q+1, status);

// Perform a combine (allreduce) operation on row members
Grid2d.Row.Allreduce (&sbuf, &rbuf, 1, MPI_INT, MPI_MAX);

// Perform a reduce operation on the PQ plane of a 3d grid
Grid3d.PQ_Plane.Allreduce (&sbuf, &rbuf, 1, MPI_INT, MPI_MAX);
```

Figure 10: An example to illustrate the use of MPIX_Grid2d and MPIX_Grid3d classes.

- the topology classes described in this paper as well as more general topologies,
- inter-communicator collective operations,
- inter-communicator servers,
- inter-communicator manipulation routines,
- experimental I/O services.

Furthermore, we anticipate addressing these issues:

- The current MPI++ interface consists of C++ wrappers around the C MPI functions. In order to exploit the power of C++ fully, we may write portions of MPI++ directly in C++. In order to do that, we may have to change the C bindings specified in the standard.
- We will explore ways of exploiting overloading that allow us to provide multiple ways of calling MPI functions. For example, almost all MPI functions return an error code even though most users will not check or care about the error code returned. It may prove useful to provide additional bindings that return more useful information (like a created object) with no error code returned.

7 Summary and Conclusions

MPI provides an environment for writing portable, efficient, and heterogeneous message-passing code in C or FORTRAN 77, but does not provide an interface to an object-oriented language such as C++. We have presented a C++ interface and discussed how ideas relating to datatypes and topologies from *Zipcode* and MPI can be combined.

The C++ interface presented here is currently distributed with the MPI model implementation by Mississippi State University and Argonne National Laboratory. This is available by anonymous ftp in the directory **pub/mpi** on *info.mcs.anl.gov*.

References

[1] A. Beguelin, G. A. Geist, W. Jiang, R. Manchek, K. Moore, and V. Sunderam. The PVM project. Technical report, Oak Ridge National Laboratory, February 1993.

[2] M.L. Branstetter, J.A. Guse, and D.M. Nessett. ELROS – An Embedded Language for Remote Operations Service. Technical Report UCRL-JC-108862, Lawrence Livermore National Laboratory, November 1991.

[3] M.L. Branstetter, J.A. Guse, D.M. Nessett, and L.C. Stanberry. An ELROS Primer. Technical report, Lawrence Livermore National Laboratory, 1992.

[4] R. Butler and E. Lusk. User's guide to the P4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992.

[5] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable Programming with the PARMACS Message–Passing Library. *Parallel Computing*, 1994.

[6] Nathan E. Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. An initial implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[7] 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, April 1992. Also available as LLNL Technical Report UCRL-JC-109775.

[8] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report No. CS-93-214., University of Tennessee, November 2, 1993. Available on **netlib**.

[9] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A Users' Guide to PICL – A Portable Instrumented Communication Library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, May 1992.

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

[11] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993.

[12] Anthony Skjellum, Alvin P. Leung, Charles H. Still Steven G. Smith, Robert D. Falgout, and Chuck H. Baldwin. The Multicomputer Toolbox – First-Generation Scalable Libraries. In *Proceedings of HICSS–27*, page in press. IEEE Computer Society Press, 1994. HICSS–27 Minitrack on Tools and Languages for Transportable Parallel Applications.

[13] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 1994. (Invited Paper, to appear).

[14] Steven G. Smith, Robert D. Falgout, Charles H. Still, and Anthony Skjellum. High-Level Message-Passing Constructs for Zipcode 1.0: Design and Implementation. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 150–159. IEEE Computer Society Press, 1993.