# A User's Guide to MPI

Peter S. Pacheco
Department of Mathematics
University of San Francisco
San Francisco, CA 94117
peter@usfca.edu

March 30, 1998

## Contents

# 1 Introduction

The Message-Passing Interface or MPI is a library of functions and macros that can be used in C, FORTRAN, and C++ programs. As its name implies, MPI is intended for use in programs that exploit the existence of multiple processors by message-passing.

MPI was developed in 1993–1994 by a group of researchers from industry, government, and academia. As such, it is one of the first standards for programming parallel processors, and it is the first that is based on message-passing.

This *User's Guide* is a brief tutorial introduction to some of the more important features of MPI for C programmers. It is intended for use by programmers who have some experience using C but little experience with message-passing. It is based on parts of the book [6], which is to be published by Morgan Kaufmann. For comprehensive guides to MPI see [4], [5] and [2]. For an extended, elementary introduction, see [6].

**Acknowledgments.** My thanks to nCUBE and the USF faculty development fund for their support of the work that went into the preparation of this *Guide.* Work on MPI was supported in part by the Advanced Research Projects Agency under contract number NSF-ASC-9310330, administered by the National Science Foundation's Division of Advanced Scientific Computing. The author gratefully acknowledges use of the Argonne High-Performance Computing Research Facility. The HPCRF is funded principally by the U.S. Department of Energy Office of Scientific Computing.

## 2 Greetings!

The first C program that most of us saw was the “Hello, world!” program in Kernighan and Ritchie’s classic text, *The C Programming Language* [3]. It simply prints the message “Hello, world!” A variant that makes some use of multiple processes is to have each process send a greeting to another process.

In MPI, the processes involved in the execution of a parallel program are identified by a sequence of non-negative integers. If there are $p$ processes executing a program, they will have ranks 0, 1, ..., $p-1$. The following program has each process other than 0 send a message to process 0, and process 0 prints out the messages it received.

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv) {
    int my_rank;        /* Rank of process */
    int p;              /* Number of processes */
    int source;         /* Rank of sender */
    int dest;           /* Rank of receiver */
    int tag = 50;       /* Tag for messages */
    char message[100];  /* Storage for the message */
    MPI_Status status;  /* Return status for receive */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen(message)+1 to include '\0' */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
            tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize();
} /* main */
```

The details of compiling and executing this program depend on the system you're using. So ask your local guide how to compile and run a parallel program that uses MPI. We discuss the freely available systems in an appendix.

When the program is compiled and run with two processes, the output should be

```
Greetings from process 1!
```

If it's run with four processes, the output should be

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```

Although the details of what happens when the program is executed vary from machine to machine, the essentials are the same on all machines, provided we run one process on each processor.

1. The user issues a directive to the operating system which has the effect of placing a copy of the executable program on each processor.

2. Each processor begins execution of its copy of the executable.

3. Different processes can execute different statements by branching within the program. Typically the branching will be based on process ranks.

So the Greetings program uses the *Single Program Multiple Data* or *SPMD* paradigm. That is, we obtain the *effect* of different programs running on different processors by taking branches within a single program on the basis of process rank: the statements executed by process 0 are different from those executed by the other processes, even though all processes are running the same program. This is the most commonly used method for writing MIMD programs, and we'll use it exclusively in this *Guide.*

## 2.1 General MPI Programs

Every MPI program must contain the preprocessor directive

```
    #include "mpi.h"
```

This file, mpi.h, contains the definitions, macros and function prototypes necessary for compiling an MPI program.

Before any other MPI functions can be called, the function MPI_Init must be called, and it should only be called once. Its arguments are pointers to the main function's parameters — argc and argv. It allows systems to do any special set-up so that the MPI library can be used. After a program has finished using the MPI library, it must call MPI_Finalize. This cleans up any "unfinished business" left by MPI — e.g., pending receives that were never completed. So a typical MPI program has the following layout.

```
         ⋮
#include "mpi.h"
         ⋮
main(int argc, char** argv) {
         ⋮
    /* No MPI functions called before this */
    MPI_Init(&argc, &argv);
         ⋮
    MPI_Finalize();
    /* No MPI functions called after this */
         ⋮
} /* main */
         ⋮
```

## 2.2 Finding Out About the Rest of the World

MPI provides the function MPI_Comm_rank, which returns the rank of a process in its second argument. Its syntax is

```
int MPI_Comm_rank(MPI_Comm comm, int rank)
```

The first argument is a *communicator*. Essentially a communicator is a collection of processes that can send messages to each other. For basic programs, the only communicator needed is MPI_COMM_WORLD. It is predefined in MPI and consists of all the processes running when program execution begins.

Many of the constructs in our programs also depend on the number of processes executing the program. So MPI provides the function MPI_Comm_size for determining this. Its first argument is a communicator. It returns the number of processes in a communicator in its second argument. Its syntax is

```
int MPI_Comm_size(MPI_Comm comm, int size)
```

## 2.3 Message: Data + Envelope

The actual message-passing in our program is carried out by the MPI functions MPI_Send and MPI_Recv. The first command sends a message to a designated process. The second receives a message from a process. These are the most basic message-passing commands in MPI. In order for the message to be successfully communicated the system must append some information to the data that the application program wishes to transmit. This additional information forms the *envelope* of the message. In MPI it contains the following information.

1. The rank of the receiver.

2. The rank of the sender.

3. A tag.

4. A communicator.

These items can be used by the receiver to distinguish among incoming messages. The *source* argument can be used to distinguish messages received from different processes. The *tag* is a user-specified int that can be used to distinguish messages received from a single process. For example, suppose process $A$ is sending two messages to process $B$; both messages contain a single float. One of the floats is to be used in a calculation, while the other is to be printed. In order to determine which is which, $A$ can use different tags for the two messages. If $B$ uses the same two tags in the corresponding receives, when it receives the messages, it will "know" what to do with them. MPI guarantees that the integers 0–32767 can be used as tags. Most implementations allow much larger values.

As we noted above, a communicator is basically a collection of processes that can send messages to each other. When two processes are communicating using MPI_Send and MPI_Receive, its importance arises when separate modules of a program have been written independently of each other. For example, suppose we wish to solve a system of differential equations, and, in the course of solving the system, we need to solve a system of linear equations. Rather than writing the linear system solver from scratch, we might want to use a *library* of functions for solving linear systems that was written by someone else and that has been highly optimized for the system we're using. How do we avoid confusing the messages *we* send from process $A$ to process $B$ with those sent by the library functions? Before the advent of communicators, we would probably have to partition the set of valid tags, setting aside some of them for exclusive use by the library functions. This is tedious and it will cause problems if we try to run our program on another system: the other system's linear solver may not (probably won't) require the same set of tags. With the advent of communicators, we simply create a communicator that can be used exclusively by the linear solver, and pass it as an argument in calls to the solver. We'll discuss the details of this later. For now, we can get away with using the predefined communicator MPI_COMM_WORLD. It consists of all the processes running the program when execution begins.

## 2.4 MPI_Send and MPI_Receive

To summarize, let's detail the syntax of MPI_Send and MPI_Receive.

```
    int MPI_Send(void* message, int count,
        MPI_Datatype datatype, int dest, int tag,
        MPI_Comm comm)

    int MPI_Recv(void* message, int count,
        MPI_Datatype datatype, int source, int tag,
        MPI_Comm comm, MPI_Status* status)
```

Like most functions in the standard C library most MPI functions return an integer error code. However, like most C programmers, we will ignore these return values in most cases.

The contents of the message are stored in a block of memory referenced by the argument message. The next two arguments, count and datatype, allow the system to identify the end of the message: it contains a sequence of count values, each having *MPI* type datatype. This type is not a C type, although most of the predefined types correspond to C types. The predefined MPI types and the corresponding C types (if they exist) are listed in the following table.

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

The last two types, MPI_BYTE and MPI_PACKED, don't correspond to standard C types. The MPI_BYTE type can be used if you wish to force the system to perform no conversion between different data representations (e.g., on a heterogeneous network of workstations using different representations of data). We'll discuss the type MPI_PACKED later.

Note that the amount of space allocated for the receiving buffer does not have to match the exact amount of space in the message being received. For example, when our program is run, the size of the message that process 1 sends, strlen(message)+1, is 28 chars, but process 0 receives the message in a buffer that has storage for 100 characters. This makes sense. In general, the receiving process may not know the exact size of the message being sent. So MPI allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage, an overflow error occurs [4].

The arguments dest and source are, respectively, the ranks of the receiving and the sending processes. MPI allows source to be a "wildcard." There is a predefined constant MPI_ANY_SOURCE that can be used if a process is ready to receive a message from *any* sending process rather than a particular sending process. There is *not* a wildcard for dest.

As we noted earlier, MPI has two mechanisms specifically designed for "partitioning the message space:" tags and communicators. The arguments tag and comm are, respectively, the tag and communicator. The tag is an int, and, for now, our only communicator is MPI_COMM_WORLD, which, as we noted earlier is predefined on all MPI systems and consists of all the processes running when execution of the program begins. There is a wildcard, MPI_ANY_TAG, that MPI_Recv can use for the tag. There is *no* wildcard for the communicator. In other words, in order for process $A$ to send a message to process $B$, the argument comm that $A$ uses in MPI_Send must be identical to the argument that $B$ uses in MPI_Recv.

The last argument of MPI_Recv, status, returns information on the data that was actually received. It references a record with with two fields — one for the source and one for the tag. So if, for example, the *source* of the receive was MPI_ANY_SOURCE, then status will contain the rank of the process that sent the message.

# 3 Collective Communication

There are probably a few things in the trapezoid rule program that we can improve on. For example, there is the I/O issue. There are also a couple of problems we haven't discussed yet. Let's look at what happens when the program is run with eight processes.

All the processes begin executing the program (more or less) simultaneously. However, after carrying out the basic set-up tasks (calls to MPI_Init, MPI_Comm_size, and MPI_Comm_rank), processes 1–7 are idle while process 0 collects the input data. We don't want to have idle processes, but in view of our restrictions on which processes can read input, there isn't much we can do about this. However, after process 0 has collected the input data, the higher rank processes must continue to wait while 0 sends the input data to the lower rank processes. This isn't just an I/O issue. Notice that there is a similar inefficiency at the end of the program, when process 0 does all the work of collecting and adding the local integrals.

Of course, this is highly undesirable: the main point of parallel processing is to get multiple processes to collaborate on solving a problem. If one of the processes is doing most of the work, we might as well use a conventional, single-processor machine.

## 3.1 Tree-Structured Communication

Let's try to improve our code. We'll begin by focussing on the distribution of the input data. How can we divide the work more evenly among the processes? A natural solution is to imagine that we have a tree of processes, with 0 at the root.

During the first stage of the data distribution, 0 sends the data to (say) 4. During the next stage, 0 sends the data to 2, while 4 sends it to 6. During the last stage, 0 sends to 1, while 2 sends to 3, 4 sends to 5, and 6 sends to 7. (See figure 3.1.) So we have reduced our input distribution loop from 7 stages to 3 stages. More generally, if we have $p$ processes, this procedure allows us to distribute the input data in $\lceil \log_2(p) \rceil$* stages, rather than $p-1$ stages, which, if $p$ is large, is a huge savings.

In order to modify the Get_data function to use a tree-structured distri-

*The notation $\lceil x \rceil$ denotes the smallest whole number greater than or equal to $x$.

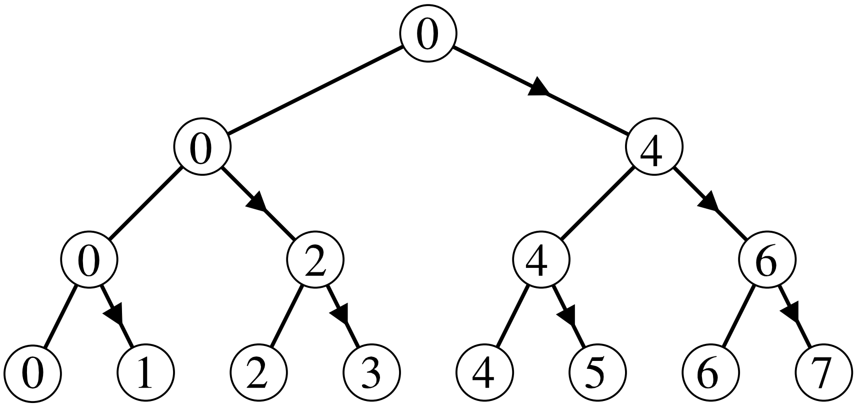

Figure 1: Processors configured as a tree

bution scheme, we need to introduce a loop with $\lceil \log_2(p) \rceil$ stages. In order to implement the loop, each process needs to calculate at each stage

- whether it receives, and, if so, the source; and
- whether it sends, and, if so, the destination.

As you can probably guess, these calculations can be a bit complicated, especially since there is no canonical choice of ordering. In our example, we chose:

1. 0 sends to 4.
2. 0 sends to 2, 4 sends to 6.
3. 0 sends to 1, 2 sends to 3, 4 sends to 5, 6 sends to 7.

We might also have chosen (for example):

1. 0 sends to 1.
2. 0 sends to 2, 1 sends to 3.
3. 0 sends to 4, 1 sends to 5, 2 sends to 6, 3 sends to 7.

Indeed, unless we know something about the underlying topology of our machine, we can't really decide which scheme is better.

So ideally we would prefer to use a function that has been specifically tailored to the machine we're using so that we won't have to worry about all these tedious details, and we won't have to modify our code every time we change machines. As you may have guessed, MPI provides such a function.

## 3.2 Broadcast

A communication pattern that involves all the processes in a communicator is a *collective communication.* As a consequence, a collective communication usually involves more than two processes. A *broadcast* is a collective communication in which a single process sends the same data to every process. In MPI the function for broadcasting data is MPI_Bcast:

```
int MPI_Bcast(void* message, int count,
    MPI_Datatype datatype, int root, MPI_Comm comm)
```

It simply sends a copy of the data in message on process root to each process in the communicator comm. It should be called by all the processes in the communicator with the same arguments for root and comm. Hence a broadcast message cannot be received with MPI_Recv. The parameters count and datatype have the same function that they have in MPI_Send and MPI_Recv: they specify the extent of the message. However, unlike the *point-to-point* functions, MPI insists that in collective communication count and datatype be the same on all the processes in the communicator [4]. The reason for this is that in some collective operations (see below), a single process will receive data from many other processes, and in order for a program to determine how much data has been received, it would need an entire *array* of return statuses.

We can rewrite the Get_data function using MPI_Bcast as follows.

```
void Get_data2(int my_rank, float* a_ptr, float* b_ptr,
    int* n_ptr) {
    int root = 0;  /* Arguments to MPI_Bcast */
    int count = 1;

    if (my_rank == 0)
    {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    MPI_Bcast(a_ptr, 1, MPI_FLOAT, root,
        MPI_COMM_WORLD);
    MPI_Bcast(b_ptr, 1, MPI_FLOAT, root,
        MPI_COMM_WORLD);
    MPI_Bcast(n_ptr, 1, MPI_INT, root,
        MPI_COMM_WORLD);
} /* Get_data2 */
```

Certainly this version of Get_data is much more compact and readily comprehensible than the original, and if MPI_Bcast has been optimized for your system, it will also be a good deal faster.

## 3.3 Reduce

In the trapezoid rule program after the input phase, every processor executes essentially the same commands until the final summation phase. So unless our function $f(x)$ is fairly complicated (i.e., it requires considerably more work to evaluate over certain parts of $[a, b]$), this part of the program distributes the work equally among the processors. As we have already noted, this is *not* the case with the final summation phase, when, once again, process 0 gets a disproportionate amount of the work. However, you have probably already noticed that by reversing the arrows in figure 3.1, we can use the same idea we used in section 3.1. That is, we can distribute the work of calculating the sum among the processors as follows.

1. (a) 1 sends to 0, 3 sends to 2, 5 sends to 4, 7 sends to 6.

   (b) 0 adds its integral to that of 1, 2 adds its integral to that of 3, etc.

2. (a) 2 sends to 0, 6 sends to 4.

   (b) 0 adds, 4 adds.

3. (a) 4 sends to 0.

   (b) 0 adds.

Of course, we run into the same question that occurred when we were writing our own broadcast: is this tree structure making optimal use of the topology of our machine? Once again, we have to answer that this depends on the machine. So, as before, we should let MPI do the work, by using an optimized function.

The "global sum" that we wish to calculate is an example of a general class of collective communication operations called *reduction operations*. In a global reduction operation, all the processes (in a communicator) contribute data which is combined using a binary operation. Typical binary operations are addition, max, min, logical and, etc. The MPI function for performing a reduction operation is

```
int MPI_Reduce(void* operand, void* result,
    int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm)
```

MPI_Reduce combines the operands stored in *operand using operation op and stores the result in *result on process root. Both operand and result refer to count memory locations with type datatype. MPI_Reduce must be called by all processes in the communicator comm, and count, datatype, and op must be the same on each process.

The argument op can take on one of the following predefined values.

| Operation Name | Meaning |
|---|---|
| MPI_MAX | Maximum |
| MPI_MIN | Minimum |
| MPI_SUM | Sum |
| MPI_PROD | Product |
| MPI_LAND | Logical And |
| MPI_BAND | Bitwise And |
| MPI_LOR | Logical Or |
| MPI_BOR | Bitwise Or |
| MPI_LXOR | Logical Exclusive Or |
| MPI_BXOR | Bitwise Exclusive Or |
| MPI_MAXLOC | Maximum and Location of Maximum |
| MPI_MINLOC | Minimum and Location of Minimum |

It is also possible to define additional operations. For details see [4].

As an example, let's rewrite the last few lines of the trapezoid rule program.

```
        ⋮
/* Add up the integrals calculated by each process */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);

/* Print the result */
        ⋮
```

Note that each processor calls MPI_Reduce with the same arguments. In particular, even though total only has significance on process 0, each process must supply an argument.

## 3.4 Other Collective Communication Functions

MPI supplies many other collective communication functions. We briefly enumerate some of these here. For full details, see [4].

-
  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  MPI_Barrier provides a mechanism for synchronizing all the processes in the communicator comm. Each process blocks (i.e., pauses) until every process in comm has called MPI_Barrier.

-
  ```
  int MPI_Gather(void* send_buf, int send_count,
      MPI_Datatype send_type,  void* recv_buf,
      int recv_count, MPI_Datatype recv_type,
      int root, MPI_comm comm)
  ```

  Each process in comm sends the contents of send_buf to the process with rank root. The process root concatenates the received data in process rank order in recv_buf. That is, the data from process 0 is followed by the data from process 1, which is followed by the data from process 2, etc. The recv arguments are significant only on the process with rank root. The argument recv_count indicates the number of items received from each process — not the total number received.

-
  ```
  int MPI_Scatter(void* send_buf, int send_count,
      MPI_Datatype send_type, void* recv_buf,
      int recv_count, , MPI_Datatype recv_type,
      int root, MPI_Comm comm)
  ```

  The process with rank root distributes the contents of send_buf among the processes. The contents of send_buf are split into $p$ segments each consisting of send_count items. The first segment goes to process 0, the second to process 1, etc. The send arguments are significant only on process root.

-
  ```
  int MPI_Allgather(void* send_buf, int send_count,
      MPI_Datatype send_type, void* recv_buf,
      int recv_count, MPI_Datatype recv_type,
      MPI_comm comm)
  ```

  MPI_Allgather gathers the contents of each send_buf on each process. Its *effect* is the same as if there were a sequence of $p$ calls to MPI_Gather, each of which has a different process acting as root.

-
  ```
  int MPI_Allreduce(void* operand, void* result,
      int count, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm)
  ```

  MPI_Allreduce stores the result of the reduce operation op in each process' result buffer.

# 4 Grouping Data for Communication

With current generation machines sending a message is an expensive operation. So as a rule of thumb, the fewer messages sent, the better the overall performance of the program. However, in each of our trapezoid rule programs, when we distributed the input data, we sent $a, b$, and $n$ in separate messages — whether we used MPI_Send and MPI_Recv or MPI_Bcast. So we should be able to improve the performance of the program by sending the three input values in a single message. MPI provides three mechanisms for grouping individual data items into a single message: the count parameter to the various communication routines, derived datatypes, and MPI_Pack/MPI_Unpack. We examine each of these options in turn.

## 4.1 The Count Parameter

Recall that MPI_Send, MPI_Receive, MPI_Bcast, and MPI_Reduce all have a count and a datatype argument. These two parameters allow the user to group data items having the same basic type into a single message. In order to use this, the grouped data items must be stored in *contiguous* memory locations. Since C guarantees that array elements are stored in contiguous memory locations, if we wish to send the elements of an array, or a subset of an array, we can do so in a single message. In fact, we've already done this in section 2, when we sent an array of char.

As another example, suppose we wish to send the second half of a vector containing 100 floats from process 0 to process 1.

```
float vector[100];
int tag, count, dest, source;
MPI_Status status;
int p;
int my_rank;
     ⋮
if (my_rank == 0) {
    /*  Initialize vector and send */
     ⋮
    tag = 47;
    count = 50;
```

```
        dest = 1;
        MPI_Send(vector + 50, count, MPI_FLOAT, dest, tag,
            MPI_COMM_WORLD);
    } else { /* my_rank == 1 */
        tag = 47;
        count = 50;
        source = 0;
        MPI_Recv(vector+50, count, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, &status);
    }
```

Unfortunately, this doesn't help us with the trapezoid rule program. The data we wish to distribute to the other processes, a, b, and n, are not stored in an array. So even if we declared them one after the other in our program,

```
    float a;
    float b;
    int n;
```

C does *not* guarantee that they are stored in contiguous memory locations. One might be tempted to store n as a float and put the three values in an array, but this would be poor programming style and it wouldn't address the fundamental issue. In order to solve the problem we need to use one of MPI's other facilities for grouping data.

## 4.2 Derived Types and MPI_Type_struct

It might seem that another option would be to store $a, b$, and $n$ in a struct with three members — two floats and an int — and try to use the datatype argument to MPI_Bcast. The difficulty here is that the type of datatype is MPI_Datatype, which is an actual type itself — not the same thing as a user-defined type in C. For example, suppose we included the type definition

```
    typedef struct {
        float a;
        float b;
        int n;
    } INDATA_TYPE
```

and the variable definition

```
    INDATA_TYPE indata
```

Now if we call MPI_Bcast

```
    MPI_Bcast(&indata, 1, INDATA_TYPE, 0, MPI_COMM_WORLD)
```

the program will fail. The details depend on the implementation of MPI that you're using. If you have an ANSI C compiler, it will flag an error in the call to MPI_Bcast, since INDATA_TYPE does not have type MPI_Datatype. The problem here is that MPI is a *pre-existing* library of functions. That is, the MPI functions were written without knowledge of the datatypes that you define in your program. In particular, none of the MPI functions "knows" about INDATA_TYPE.

MPI provides a partial solution to this problem, by allowing the user to build MPI datatypes at execution time. In order to build an MPI datatype, one essentially specifies the layout of the data in the type — the member types and their relative locations in memory. Such a type is called a *derived datatype.* In order to see how this works, let's write a function that will build a derived type that corresponds to INDATA_TYPE.

```
void Build_derived_type(INDATA_TYPE* indata,
    MPI_Datatype* message_type_ptr){

    int block_lengths[3];
    MPI_Aint displacements[3];
    MPI_Aint addresses[4];
    MPI_Datatype typelist[3];

    /*  Build a derived datatype consisting of
     *  two floats and an int */

    /* First specify the types */
    typelist[0] = MPI_FLOAT;
    typelist[1] = MPI_FLOAT;
    typelist[2] = MPI_INT;

    /* Specify the number of elements of each type */
    block_lengths[0] = block_lengths[1] =
        block_lengths[2] = 1;

    /* Calculate the displacements of the members
     * relative to indata */
    MPI_Address(indata, &addresses[0]);
    MPI_Address(&(indata->a), &addresses[1]);
    MPI_Address(&(indata->b), &addresses[2]);
    MPI_Address(&(indata->n), &addresses[3]);
    displacements[0] = addresses[1] - addresses[0];
    displacements[1] = addresses[2] - addresses[0];
    displacements[2] = addresses[3] - addresses[0];

    /* Create the derived type */
    MPI_Type_struct(3, block_lengths, displacements, typelist,
         message_type_ptr);

    /* Commit it so that it can be used */
    MPI_Type_commit(message_type_ptr);
} /* Build_derived_type */
```

The first three statements specify the types of the members of the derived type, and the next specifies the number of elements of each type. The next four calculate the addresses of the three members of indata. The next three statements use the calculated addresses to determine the *displacements* of the three members relative to the address of the first — which is given displacement 0. With this information, we know the types, sizes and relative locations of the members of a variable having C type INDATA_TYPE, and hence we can define a derived data type that corresponds to the C type. This is done by calling the functions MPI_Type_struct and MPI_Type_commit.

The newly created MPI datatype can be used in any of the MPI communication functions. In order to use it, we simply use the starting address of a variable of type INDATA_TYPE as the first argument, and the derived type in the datatype argument. For example, we could rewrite the Get_data function as follows.

```
void Get_data3(INDATA_TYPE* indata, int my_rank){
    MPI_Datatype message_type;  /* Arguments to */
    int root = 0;               /* MPI_Bcast */
    int count = 1;

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d",
            &(indata->a), &(indata->b), &(indata->n));
    }

    Build_derived_type(indata, &message_type);
    MPI_Bcast(indata, count, message_type, root,
        MPI_COMM_WORLD);
} /* Get_data3 */
```

A few observations are in order. Note that we calculated the addresses of the members of indata with MPI_Address rather than C's & operator. The reason for this is that ANSI C does not require that a pointer be an int (although this is commonly the case). See [4], for a more detailed discussion of this point. Note also that the type of array_of_displacements is MPI_Aint — not int. This is a special type in MPI. It allows for the possibility that addresses are too large to be stored in an int.

To summarize, then, we can build general derived datatypes by calling MPI_Type_struct. The syntax is

```
    int MPI_Type_Struct(int count,
        int* array_of_block_lengths,
        MPI_Aint* array_of_displacements,
        MPI_Datatype* array_of_types,
        MPI_Datatype* newtype)
```

The argument count is the number of elements in the derived type. It is also the size of the three arrays, array_of_block_lengths, array_of_displacements, and array_of_types. The array array_of_block_lengths contains the number of entries in each element of the type. So if an element of the type is an array of $m$ values, then the corresponding entry in array_of_block_lengths is $m$. The array array_of_displacements contains the displacement of each element from the beginning of the message, and the array array_of_types contains the MPI datatype of each entry. The argument newtype returns a pointer to the MPI datatype created by the call to MPI_Type_struct.

Note also that newtype and the entries in array_of_types all have type MPI_Datatype. So MPI_Type_struct can be called recursively to build more complex derived datatypes.

## 4.3 Other Derived Datatype Constructors

MPI_Type_struct is the most general datatype constructor in MPI, and as a consequence, the user must provide a *complete* description of each element of the type. If the data to be transmitted consists of a subset of the entries in an array, we shouldn't need to provide such detailed information, since all the elements have the same basic type. MPI provides three derived datatype constructors for dealing with this situation: MPI_Type_Contiguous, MPI_Type_vector and MPI_Type_indexed. The first constructor builds a derived type whose elements are contiguous entries in an array. The second builds a type whose elements are equally spaced entries of an array, and the third builds a type whose elements are arbitrary entries of an array. Note that before any derived type can be used in communication it must be *committed* with a call to MPI_Type_commit.

Details of the syntax of the additional type constructors follow.

-
  ```
  int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
      MPI_Datatype* newtype)
  ```

  MPI_Type_contiguous creates a derived datatype consisting of count elements of type oldtype. The elements belong to contiguous memory locations.

-
  ```
  int MPI_Type_vector(int count, int block_length,
          int stride, MPI_Datatype element_type,
          MPI_Datatype* newtype)
  ```

  MPI_Type_vector creates a derived type consisting of count elements. Each element contains block_length entries of type element_type. Stride is the number of elements of type element_type between successive elements of new_type.

-
  ```
  int MPI_Type_indexed(int count,
      int* array_of_block_lengths,
      int* array_of_displacements,
      MPI_Datatype element_type,
      MPI_Datatype* newtype)
  ```

  MPI_Type_indexed creates a derived type consisting of count elements. The $i$th element $(i = 0, 1, \ldots, \mathsf{count} - 1)$, consists of array_of_block_lengths[i] entries of type element_type, and it is displaced array_of_displacements[i] units of type element_type from the beginning of newtype.

## 4.4 Pack/Unpack

An alternative approach to grouping data is provided by the MPI functions MPI_Pack and MPI_Unpack. MPI_Pack allows one to explicitly store noncontiguous data in contiguous memory locations, and MPI_Unpack can be used to copy data from a contiguous buffer into noncontiguous memory locations. In order to see how they are used, let's rewrite Get_data one last time.

```
void Get_data4(int my_rank, float* a_ptr, float* b_ptr,
    int* n_ptr) {
    int root = 0;      /* Argument to MPI_Bcast */
    char buffer[100];  /* Arguments to MPI_Pack/Unpack */
    int position;      /* and MPI_Bcast*/

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

        /* Now pack the data into buffer */
        position = 0;  /* Start at beginning of buffer */
        MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented by */
        /* sizeof(float) bytes */
        MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100,
```

```
            &position, MPI_COMM_WORLD);
        MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100,
            &position, MPI_COMM_WORLD);

        /* Now broadcast contents of buffer */
        MPI_Bcast(buffer, 100, MPI_PACKED, root,
            MPI_COMM_WORLD);
    } else {
        MPI_Bcast(buffer, 100, MPI_PACKED, root,
            MPI_COMM_WORLD);

        /* Now unpack the contents of buffer */
        position = 0;
        MPI_Unpack(buffer, 100, &position, a_ptr, 1,
            MPI_FLOAT, MPI_COMM_WORLD);
        /* Once again position has been incremented */
        /* by sizeof(float) bytes */
        MPI_Unpack(buffer, 100, &position, b_ptr, 1,
            MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, 100, &position, n_ptr, 1,
            MPI_INT, MPI_COMM_WORLD);
    }
} /* Get_data4 */
```

In this version of Get_data process 0 uses MPI_Pack to copy a to buffer and then append b and n. After the broadcast of buffer, the remaining processes use MPI_Unpack to successively extract a, b, and n from buffer. Note that the datatype for the calls to MPI_Bcast is MPI_PACKED.

The syntax of MPI_Pack is

```
    int MPI_Pack(void* pack_data, int in_count,
        MPI_Datatype datatype, void* buffer,
        int size, int* position_ptr, MPI_Comm comm)
```

The parameter pack_data references the data to be buffered. It should consist of in_count elements, each having type datatype. The parameter position_ptr is an *in/out* parameter. On input, the data referenced by pack_data is copied into memory starting at address buffer + *position_ptr. On return, *position_ptr references the first location in buffer *after* the data that was copied. The parameter size contains the size *in bytes* of the memory referenced by buffer, and comm is the communicator that will be using buffer.

The syntax of MPI_Unpack is

```
int MPI_Unpack(void* buffer, int size,
    int* position_ptr, void* unpack_data, int count,
    MPI_Datatype datatype, MPI_comm comm)
```

The parameter buffer references the data to be unpacked. It consists of size bytes. The parameter position_ptr is once again an *in/out* parameter. When MPI_Unpack is called, the data starting at address buffer + *position_ptr is copied into the memory referenced by unpack_data. On return, *position_ptr references the first location in buffer after the data that was just copied. MPI_Unpack will copy count elements having type datatype into unpack_data. The communicator associated with buffer is comm.

## 4.5 Deciding Which Method to Use

If the data to be sent is stored in consecutive entries of an array, then one should simply use the count and datatype arguments to the communication function(s). This approach involves no additional overhead in the form of calls to derived datatype creation functions or calls to MPI_Pack/MPI_Unpack.

If there are a large number of elements that are not in contiguous memory locations, then building a derived type will probably involve less overhead than a large number of calls to MPI_Pack/MPI_Unpack.

If the data all have the same type and are stored at regular intervals in memory (e.g., a column of a matrix), then it will almost certainly be much easier and faster to use a derived datatype than it will be to use MPI_Pack/MPI_Unpack. Furthermore, if the data all have the same type, but are stored in irregularly spaced locations in memory, it will still probably be easier and more efficient to create a derived type using MPI_Type_indexed. Finally, if the data are heterogeneous and one is repeatedly sending the same collection of data (e.g., row number, column number, matrix entry), then it will be better to use a derived type, since the overhead of creating the derived type is incurred only once, while the overhead of calling

MPI_Pack/MPI_Unpack must be incurred every time the data is communicated.

This leaves the case where one is sending heterogeneous data only once, or very few times. In this case, it may be a good idea to collect some information on the cost of derived type creation and packing/unpacking the data. For example, on an nCUBE 2 running the MPICH implementation of MPI, it takes about 12 milliseconds to create the derived type used in Get_data3, while it only takes about 2 milliseconds to pack or unpack the data in Get_data4. Of course, the saving isn't as great as it seems because of the asymmetry in the pack/unpack procedure. That is, while process 0 packs the data, the other processes are idle, and the entire function won't complete until both the pack and unpack are executed. So the cost ratio is probably more like 3:1 than 6:1.

There are also a couple of situations in which the use of MPI_Pack and MPI_Unpack is preferred. Note first that it may be possible to avoid the use of *system* buffering with pack, since the data is explicitly stored in a user-defined buffer. The system can exploit this by noting that the message datatype is MPI_PACKED. Also note that the user can send "variable-length" messages by packing the number of elements at the beginning of the buffer. For example, suppose we want to send rows of a sparse matrix. If we have stored a row as a pair of arrays — one containing the column subscripts, and one containing the corresponding matrix entries — we could send a row from process 0 to process 1 as follows.

```
float* entries;
int* column_subscripts;
int nonzeroes;  /* number of nonzeroes in row */
int position;
int row_number;
char* buffer[HUGE]; /* HUGE is a predefined constant */
MPI_Status status;
     ⋮
if (my_rank == 0) {
    /* Get the number of nonzeros in the row. */
    /* Allocate storage for the row. */
    /* Initialize entries and column_subscripts */
     ⋮
    /* Now pack the data and send */
    position = 0;
    MPI_Pack(&nonzeroes, 1, MPI_INT, buffer, HUGE,
        &position, MPI_COMM_WORLD);
    MPI_Pack(&row_number, 1, MPI_INT, buffer, HUGE,
        &position, MPI_COMM_WORLD);
    MPI_Pack(entries, nonzeroes, MPI_FLOAT, buffer,
        HUGE, &position, MPI_COMM_WORLD);
    MPI_Pack(column_subscripts, nonzeroes, MPI_INT,
        buffer, HUGE, &position, MPI_COMM_WORLD);
    MPI_Send(buffer, position, MPI_PACKED, 1, 193,
        MPI_COMM_WORLD);
} else { /* my_rank == 1 */
    MPI_Recv(buffer, HUGE, MPI_PACKED, 0, 193,
        MPI_COMM_WORLD, &status);
    position = 0;
    MPI_Unpack(buffer, HUGE, &position, &nonzeroes,
        1, MPI_INT, MPI_COMM_WORLD);
    MPI_Unpack(buffer, HUGE, &position, &row_number,
        1, MPI_INT, MPI_COMM_WORLD);
    /* Allocate storage for entries and column_subscripts */
    entries = (float *) malloc(nonzeroes*sizeof(float));
    column_subscripts = (int *) malloc(nonzeroes*sizeof(int));
    MPI_Unpack(buffer,HUGE, &position, entries,
        nonzeroes, MPI_FLOAT, MPI_COMM_WORLD);
    MPI_Unpack(buffer, HUGE, &position, column_subscripts,
        nonzeroes, MPI_INT, MPI_COMM_WORLD);
}
```

# 5 Communicators and Topologies

The use of communicators and topologies makes MPI different from most other message-passing systems. Recollect that, loosely speaking, a communicator is a collection of processes that can send messages to each other. A topology is a structure imposed on the processes in a communicator that allows the processes to be addressed in different ways. In order to illustrate these ideas, we will develop code to implement Fox's algorithm [1] for multiplying two square matrices.

## 5.1 Fox's Algorithm

We assume that the factor matrices $A = (a_{ij})$ and $B = (b_{ij})$ have order $n$. We also assume that the number of processes, $p$, is a perfect square, whose square root evenly divides $n$. Say $p = q^2$, and $\bar{n} = n/q$. In Fox's algorithm the factor matrices are partitioned among the processes in a *block checkerboard* fashion. So we view our processes as a virtual two-dimensional $q \times q$ grid, and each process is assigned an $\bar{n} \times \bar{n}$ submatrix of each of the factor matrices. More formally, we have a mapping

$$\phi : \{0, 1, \ldots, p-1\} \longrightarrow \{(s,t) : 0 \le s,t \le q-1\}$$

that is both one-to-one and onto. This defines our grid of processes: process $i$ belongs to the row and column given by $\phi(i)$. Further, the process with rank $\phi^{-1}(s,t)$ is assigned the submatrices

$$A_{st} = \begin{pmatrix} a_{s*\bar{n},t*\bar{n}} & \cdots & a_{(s+1)*\bar{n}-1,t*\bar{n}} \\ \vdots & & \vdots \\ a_{s*\bar{n},(t+1)*\bar{n}-1} & \cdots & a_{(s+1)*\bar{n}-1,(t+1)*\bar{n}-1} \end{pmatrix},$$

and

$$B_{st} = \begin{pmatrix} b_{s*\bar{n},t*\bar{n}} & \cdots & b_{(s+1)*\bar{n}-1,t*\bar{n}} \\ \vdots & & \vdots \\ b_{s*\bar{n},(t+1)*\bar{n}-1} & \cdots & b_{(s+1)*\bar{n}-1,(t+1)*\bar{n}-1} \end{pmatrix}.$$

For example, if $p = 9, \phi(x) = (x/3, x \bmod 3)$, and $n = 6$, then $A$ would be partitioned as follows.

| | | |
|---|---|---|
| Process 0<br>$A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$ | Process 1<br>$A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$ | Process 2<br>$A_{02} = \begin{pmatrix} a_{04} & a_{05} \\ a_{14} & a_{15} \end{pmatrix}$ |
| Process 3<br>$A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$ | Process 4<br>$A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$ | Process 5<br>$A_{12} = \begin{pmatrix} a_{24} & a_{25} \\ a_{34} & a_{35} \end{pmatrix}$. |
| Process 6<br>$A_{20} = \begin{pmatrix} a_{40} & a_{41} \\ a_{50} & a_{51} \end{pmatrix}$ | Process 7<br>$A_{21} = \begin{pmatrix} a_{42} & a_{43} \\ a_{52} & a_{53} \end{pmatrix}$ | Process 8<br>$A_{22} = \begin{pmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{pmatrix}$ |

In Fox's algorithm, the block submatrices, $A_{rs}$ and $B_{st}$, $s = 0, 1, \ldots, q-1$, are multiplied and accumulated on process $\phi^{-1}(r, t)$. The basic algorithm is:

```
for(step = 0; step < q; step++) {
    1.  Choose a submatrix of A from each row of processes.
    2.  In each row of processes broadcast the submatrix
        chosen in that row to the other processes in
        that row.
    3.  On each process, multiply the newly received
        submatrix of A by the submatrix of B currently
        residing on the process.
    4.  On each process, send the submatrix of B to the
        process directly above.  (On processes in the
        first row, send the submatrix to the last row.)
}
```

The submatrix chosen in the $r$th row is $A_{r,u}$, where

$$u = (r + \mathsf{step}) \bmod q.$$

## 5.2 Communicators

If we try to implement Fox's algorithm, it becomes apparent that our work will be greatly facilitated if we can treat certain subsets of processes as a communication universe — at least on a temporary basis. For example, in the pseudo-code

```
2.  In each row of processes broadcast the submatrix
    chosen in that row to the other processes in
    that row,
```

it would be useful to treat each row of processes as a communication universe, while in the statement

```
4.  On each process, send the submatrix of B to the
    process directly above.  (On processes in the
    first row, send the submatrix to the last row.)
```

it would be useful to treat each column of processes as a communication universe.

The mechanism that MPI provides for treating a subset of processes as a "communication" universe is the *communicator.* Up to now, we've been loosely defining a communicator as a collection of processes that can send messages to each other. However, now that we want to construct our own communicators, we will need a more careful discussion.

In MPI, there are two types of communicators: *intra-communicators* and *inter-communicators.* Intra-communicators are essentially a collection of processes that can send messages to each other *and* engage in collective communication operations. For example, MPI_COMM_WORLD is an intra-communicator, and we would like for each row and each column of processes in Fox's algorithm to form an intra-communicator. Inter-communicators, as the name implies, are used for sending messages between processes belonging to *disjoint* intra-communicators. For example, an inter-communicator would be useful in an environment that allowed one to dynamically create processes: a newly created set of processes that formed an intra-communicator could be linked to the original set of processes (e.g., MPI_COMM_WORLD) by an inter-communicator. We will only discuss intra-communicators. The interested reader is referred to [4] for details on the use of inter-communicators.

A minimal (intra-)communicator is composed of

- a *Group*, and
- a *Context.*

A group is an ordered collection of processes. If a group consists of $p$ processes, each process in the group is assigned a unique *rank,* which is just a nonnegative integer in the range $0, 1, \ldots, p-1$. A context can be thought of as a system-defined tag that is attached to a group. So two processes that belong to the same group and that use the same context can communicate. This pairing of a group with a context is the most basic form of a communicator. Other data can be associated to a communicator. In particular, a structure or topology can be imposed on the processes in a communicator, allowing a more natural addressing scheme. We'll discuss topologies in section 5.5.

## 5.3 Working with Groups, Contexts, and Communicators

To illustrate the basics of working with communicators, let's create a communicator whose underlying group consists of the processes in the first row of our virtual grid. Suppose that MPI_COMM_WORLD consists of $p$ processes, where $q^2 = p$. Let's also suppose that $\phi(x) = (x/q, x \bmod q)$. So the first row of processes consists of the processes with ranks 0, 1, ..., $q-1$. (Here, the ranks are in MPI_COMM_WORLD.) In order to create the group of our new communicator, we can execute the following code.

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group first_row_group;
MPI_Comm first_row_comm;
int row_size;
int* process_ranks;

/* Make a list of the processes in the new
 * communicator */
process_ranks = (int*) malloc(q*sizeof(int));
for (proc = 0; proc < q; proc++)
    process_ranks[proc] = proc;

/* Get the group underlying MPI_COMM_WORLD */
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

/* Create the new group */
MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks,
        &first_row_group);

    /* Create the new communicator */
    MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
        &first_row_comm);
```

This code proceeds in a fairly straightforward fashion to build the new communicator. First it creates a list of the processes to be assigned to the new communicator. Then it creates a group consisting of these processes. This required two commands: first get the group associated with MPI_COMM_WORLD, since this is the group from which the processes in the new group will be taken; then create the group with MPI_Group_incl. Finally, the actual communicator is created with a call to MPI_Comm_create. The call to MPI_Comm_create implicitly associates a context with the new group. The result is the communicator first_row_comm. Now the processes in first_row_comm can perform collective communication operations. For example, process 0 (in first_row_group) can broadcast $A_{00}$ to the other processes in first_row_group.

```
    int my_rank_in_first_row;
    float* A_00;

    /* my_rank is process rank in MPI_GROUP_WORLD */
    if (my_rank < q) {
        MPI_Comm_rank(first_row_comm,
            &my_rank_in_first_row);
        /* Allocate space for A_00, order = n_bar */
        A_00 = (float*) malloc (n_bar*n_bar*sizeof(float));
        if (my_rank_in_first_row == 0) {
            /* Initialize A_00 */
          ⋮
        }
        MPI_Bcast(A_00, n_bar*n_bar, MPI_FLOAT, 0,
            first_row_comm);
    }
```

Groups and communicators are *opaque objects.* From a practical standpoint, this means that the details of their internal representation depend on the particular implementation of MPI, and, as a consequence, they cannot be directly accessed by the user. Rather the user accesses a *handle* that references the opaque object, and the opaque objects are manipulated by special MPI functions, for example, MPI_Comm_create, MPI_Group_incl, and MPI_Comm_group.

Contexts are not explicitly used in any MPI functions. Rather they are implicitly associated with groups when communicators are created.

The syntax of the commands we used to create first_row_comm is fairly self-explanatory. The first command

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group* group)
```

simply returns the group underlying the communicator comm.

The second command

```
int MPI_Group_incl(MPI_Group old_group, int new_group_size,
    int* ranks_in_old_group, MPI_Group* new_group)
```

creates a new group from a list of processes in the existing group old_group. The number of processes in the new group is new_group_size, and the processes to be included are listed in ranks_in_old_group. Process 0 in new_group has rank ranks_in_old_group[0] in old_group, process 1 in new_group has rank ranks_in_old_group[1] in old_group, etc.

The final command

```
int MPI_Comm_create(MPI_Comm old_comm, MPI_Group new_group,
    MPI_Comm* new_comm)
```

associates a context with the group new_group and creates the communicator new_comm. All of the processes in new_group belong to the group underlying old_comm.

There is an extremely important distinction between the first two functions and the third. MPI_Comm_group and MPI_Group_incl, are both *local* operations. That is, there is *no* communication among processes involved in their execution. However, MPI_Comm_create *is* a collective operation. *All* the processes in old_comm must call MPI_Comm_create with the same arguments. The *Standard* [4] gives three reasons for this:

1. It allows the implementation to layer MPI_Comm_create on top of regular collective communications.

2. It provides additional safety.
3. It permits implementations to avoid communication related to context creation.

Note that since MPI_Comm_create is collective, it will behave, in terms of the data transmitted, as if it synchronizes. In particular, if several communicators are being created, they must be created in the same order on all the processes.

## 5.4 MPI_Comm_split

In our matrix multiplication program we need to create multiple communicators — one for each row of processes and one for each column. This would be an extremely tedious process if $p$ were large and we had to create each communicator using the three functions discussed in the previous section. Fortunately, MPI provides a function, MPI_Comm_split that can create several communicators simultaneously. As an example of its use, we'll create one communicator for each row of processes.

```
MPI_Comm my_row_comm;
int my_row;

/* my_rank is rank in MPI_COMM_WORLD.
 * q*q = p */
my_row = my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank,
    &my_row_comm);
```

The single call to MPI_Comm_split creates $q$ new communicators, all of them having the same name, my_row_comm. For example, if $p = 9$, the group underlying my_row_comm will consist of the processes 0, 1, and 2 on processes 0, 1, and 2. On processes 3, 4, and 5, the group underlying my_row_comm will consist of the processes 3, 4, and 5, and on processes 6, 7, and 8 it will consist of processes 6, 7, and 8.

The syntax of MPI_Comm_split is

```
int MPI_Comm_split(MPI_Comm old_comm, int split_key,
    int rank_key, MPI_Comm* new_comm)
```

It creates a new communicator for each value of split_key. Processes with the same value of split_key form a new group. The rank in the new group is determined by the value of rank_key. If process $A$ and process $B$ call MPI_Comm_split with the same value of split_key, and the rank_key argument passed by process $A$ is less than that passed by process $B$, then the rank of $A$ in the group underlying new_comm will be less than the rank of process $B$. If they call the function with the same value of rank_key, the system will arbitrarily assign one of the processes a lower rank.

MPI_Comm_split is a collective call, and it must be called by all the processes in old_comm. The function can be used even if the user doesn't wish to assign every process to a new communicator. This can be accomplished by passing the predefined constant MPI_UNDEFINED as the split_key argument. Processes doing this will have the predefined value MPI_COMM_NULL returned in new_comm.

## 5.5 Topologies

Recollect that it is possible to associate additional information — information beyond the group and context — with a communicator. This additional information is said to be *cached* with the communicator, and one of the most important pieces of information that can be cached with a communicator is a topology. In MPI, a *topology* is just a mechanism for associating different addressing schemes with the processes belonging to a group. Note that MPI topologies are *virtual* topologies — there may be no simple relation between the process structure defined by a virtual topology, and the actual underlying physical structure of the parallel machine.

There are essentially two types of virtual topologies that can be created in MPI — a *cartesian* or *grid* topology and a *graph* topology. Conceptually, the former is subsumed by the latter. However, because of the importance of grids in applications, there is a separate collection of functions in MPI whose purpose is the manipulation of virtual grids.

In Fox's algorithm we wish to identify the processes in MPI_COMM_WORLD with the coordinates of a square grid, and each row and each column of the grid needs to form its own communicator. Let's look at one method for building this structure.

We begin by associating a square grid structure with MPI_COMM_WORLD. In order to do this we need to specify the following information.

1. The number of dimensions in the grid. We have 2.

2. The size of each dimension. In our case, this is just the number of rows and the number of columns. We have $q$ rows and $q$ columns.

3. The periodicity of each dimension. In our case, this information specifies whether the first entry in each row or column is "adjacent" to the last entry in that row or column, respectively. Since we want a "circular" shift of the submatrices in each column, we want the second dimension to be periodic. It's unimportant whether the first dimension is periodic.

4. Finally, MPI gives the user the option of allowing the system to optimize the mapping of the grid of processes to the underlying physical processors by possibly reordering the processes in the group underlying the communicator. Since we don't need to preserve the ordering of the processes in MPI_COMM_WORLD, we should allow the system to reorder.

Having made all these decisions, we simply execute the following code.

```
MPI_Comm grid_comm;
int dimensions[2];
int wrap_around[2];
int reorder = 1;

dimensions[0] = dimensions[1] = q;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
    wrap_around, reorder, &grid_comm);
```

After executing this code, the communicator grid_comm will contain all the processes in MPI_COMM_WORLD (possibly reordered), and it will have a two-dimensional cartesian coordinate system associated. In order for a process to determine its coordinates, it simply calls the function MPI_Cart_coords:

```
int coordinates[2];
int my_grid_rank;
```

```
MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank, 2,
    coordinates);
```

Notice that we needed to call MPI_Comm_rank in order to get the process rank in grid_comm. This was necessary because in our call to MPI_Cart_create we set the reorder flag to 1, and hence the original process ranking in MPI_COMM_WORLD may have been changed in grid_comm.

The "inverse" to MPI_Cart_coords is MPI_Cart_rank.

```
int MPI_Cart_rank(grid_comm, coordinates,
    &grid_rank)
```

Given the coordinates of a process, MPI_Cart_rank returns the rank of the process in its third parameter process_rank.

The syntax of MPI_Cart_create is

```
int MPI_Cart_create(MPI_Comm old_comm,
    int number_of_dims, int* dim_sizes, int* periods,
    int reorder, MPI_Comm* cart_comm)
```

MPI_Cart_create creates a new communicator, cart_comm by caching a cartesian topology with old_comm. Information on the structure of the cartesian topology is contained in the parameters number_of_dims, dim_sizes, and periods. The first of these, number_of_dims, contains the number of dimensions in the cartesian coordinate system. The next two, dim_sizes and periods, are arrays with order equal to number_of_dims. The array dim_sizes specifies the order of each dimension, and periods specifies whether each dimension is circular or linear.

The processes in cart_comm are ranked in *row-major* order. That is, the first row consists of processes $0, 1, \ldots, \mathrm{dim\_sizes}[0]-1$, the second row consists of processes $\mathrm{dim\_sizes}[0], \mathrm{dim\_sizes}[0]+1, \ldots, 2*\mathrm{dim\_sizes}[0]-1$, etc. Thus it may be advantageous to change the relative ranking of the processes in old_comm. For example, suppose the *physical* topology is a $3 \times 3$ grid, and the processes (numbers) in old_comm are assigned to the processors (grid squares) as follows.

| | | |
|---|---|---|
| 3 | 4 | 5 |
| 0 | 1 | 2 |
| 6 | 7 | 8 |

Clearly, the performance of Fox's algorithm would be improved if we renumbered the processes. However, since the user doesn't know what the exact mapping of processes to processors is, we must let the system do it by setting the reorder parameter to 1.

Since MPI_Cart_create constructs a new communicator, it is a collective operation.

The syntax of the address information functions is

```
int MPI_Cart_rank(MPI_Comm comm, int* coordinates,
    int* rank);
int MPI_Cart_coords(MPI_Comm comm, int rank,
    int number_of_dims, int* coordinates)
```

MPI_Cart_rank returns the rank in the cartesian communicator comm of the process with cartesian coordinates coordinates. So coordinates is an array with order equal to the number of dimensions in the cartesian topology associated with comm. MPI_Cart_coords is the inverse to MPI_Cart_rank: it returns the coordinates of the process with rank rank in the cartesian communicator comm. Note that both of these functions are local.

## 5.6 MPI_Cart_sub

We can also partition a grid into grids of lower dimension. For example, we can create a communicator for each row of the grid as follows.

```
int varying_coords[2];
MPI_Comm row_comm;

varying_coords[0] = 0; varying_coords[1] = 1;
MPI_Cart_sub(grid_comm, varying_coords, &row_comm);
```

The call to MPI_Cart_sub creates $q$ new communicators. The varying_coords argument is an array of boolean. It specifies whether each dimension "belongs" to the new communicator. Since we're creating communicators for the rows of the grid, each new communicator consists of the processes obtained by fixing the row coordinate and letting the column coordinate *vary.* Hence we assigned varying_coords[0] the value 0 — the first coordinate doesn't vary — and we assigned varying_coords[1] the value 1 — the second coordinate varies. On each process, the new communicator is returned in row_comm. In order to create the communicators for the columns, we simply reverse the assignments to the entries in varying_coords.

```
    MPI_Comm col_comm;

    varying_coords[0] = 1; varying_coords[1] = 0;
    MPI_Cart_sub(grid_comm, varying_coord, col_comm);
```

Note the similarity of MPI_Cart_sub to MPI_Comm_split. They perform similar functions — they both partition a communicator into a collection of new communicators. However, MPI_Cart_sub can only be used with a communicator that has an associated cartesian topology, and the new communicators can only be created by fixing (or varying) one or more dimensions of the old communicators. Also note that MPI_Cart_sub is, like MPI_Comm_split, a collective operation.

## 5.7 Implementation of Fox's Algorithm

To complete our discussion, let's write the code to implement Fox's algorithm. First, we'll write a function that creates the various communicators and associated information. Since this requires a large number of variables, and we'll be using this information in other functions, we'll put it into a struct to facilitate passing it among the various functions.

```
typedef struct {
    int p;             /* Total number of processes */
    MPI_Comm comm;     /* Communicator for entire grid */
    MPI_Comm row_comm; /* Communicator for my row */
    MPI_Comm col_comm; /* Communicator for my col */
    int q;             /* Order of grid */
    int my_row;        /* My row number */
    int my_col;        /* My column number */
    int my_rank;       /* My rank in the grid communicator */
} GRID_INFO_TYPE;

/* We assume space for grid has been allocated in the
 * calling routine.
 */
void Setup_grid(GRID_INFO_TYPE* grid) {
    int old_rank;
    int dimensions[2];
    int periods[2];
    int coordinates[2];
    int varying_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;
    periods[0] = periods[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods,
        1, &(grid->comm));
    MPI_Comm_rank(grid->comm, &(grid->my_rank));
    MPI_Cart_coords(grid->comm, grid->my_rank, 2,
        coordinates);
    grid->my_row = coordinates[0];
    grid->my_col = coordinates[1];

    /* Set up row and column communicators */
    varying_coords[0] = 0; varying_coords[1] = 1;
    MPI_Cart_sub(grid->comm, varying_coords,
        &(grid->row_comm));
    varying_coords[0] = 1; varying_coords[1] = 0;
    MPI_Cart_sub(grid->comm, varying_coords,
        &(grid->col_comm));
} /* Setup_grid */
```

Notice that since each of our communicators has an associated topology, we constructed them using the topology construction functions — MPI_Cart_create and MPI_Cart_sub — rather than the more general communicator construction functions MPI_Comm_create and MPI_Comm_split.

Now let's write the function that does the actual multiplication. We'll assume that the user has supplied the type definitions and functions for the local matrices. Specifically, we'll assume she has supplied a type definition for LOCAL_MATRIX_TYPE, a corresponding derived type, DERIVED_LOCAL_MATRIX, and three functions: Local_matrix_multiply, Local_matrix_allocate, and Set_to_zero. We also assume that storage for the parameters has been allocated in the calling function, and all the parameters, except the product matrix local_C, have been initialized.

```
void Fox(int n, GRID_INFO_TYPE* grid,
    LOCAL_MATRIX_TYPE* local_A,
    LOCAL_MATRIX_TYPE* local_B,
    LOCAL_MATRIX_TYPE* local_C) {
    LOCAL_MATRIX_TYPE* temp_A;
    int step;
    int bcast_root;
    int n_bar;  /* order of block submatrix = n/q */
    int source;
    int dest;
    int tag = 43;
    MPI_Status status;

    n_bar = n/grid->q;
    Set_to_zero(local_C);

    /* Calculate addresses for circular shift of B */
    source = (grid->my_row + 1) % grid->q;
    dest = (grid->my_row + grid->q - 1) % grid->q;

    /* Set aside storage for the broadcast block of A */
    temp_A = Local_matrix_allocate(n_bar);

    for (step = 0; step < grid->q; step++) {
        bcast_root = (grid->my_row + step) % grid->q;
        if (bcast_root == grid->my_col) {
            MPI_Bcast(local_A, 1, DERIVED_LOCAL_MATRIX,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(local_A, local_B,
                local_C);
        } else {
            MPI_Bcast(temp_A, 1, DERIVED_LOCAL_MATRIX,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(temp_A, local_B,
                local_C);
        }
        MPI_Send(local_B, 1, DERIVED_LOCAL_MATRIX, dest, tag,
            grid->col_comm);
        MPI_Recv(local_B, 1, DERIVED_LOCAL_MATRIX, source, tag,
            grid->col_comm, &status);
    } /* for */

} /* Fox */
```

# 6 Where To Go From Here

## 6.1 What We Haven't Discussed

MPI is a large library. The *Standard* [4] is over 200 pages long and it defines more than 125 functions. As a consequence, this *Guide* has covered only a small fraction of MPI, and many readers will fail to find a discussion of functions that they would find very useful in their applications. So we briefly list some of the more important ideas in MPI that we have not discussed here.

1. **Communication Modes.** We have used only the *standard* communication mode for send. This means that it is up to the system to decide whether the message is buffered. MPI provides three other communication modes: *buffered, synchronous,* and *ready.* In buffered mode, the user explicitly controls the buffering of outgoing messages. In synchronous mode, a send will not complete until a matching receive is posted. In ready mode, a send may be started only if a matching receive has already been posted. MPI provides three additional send functions for these modes.

2. **Nonblocking Communication.** We have used only *blocking* sends and receives (MPI_Send and MPI_Recv.) For the send, this means that the call won't return until the message data and envelope have been buffered or sent — i.e., until the memory referenced in the call to MPI_Send is available for re-use. For the receive, this means that the call won't return until the data has been received into the memory referenced in the call to MPI_Recv. Many applications can improve their performance by using *nonblocking* communication. This means that the calls to send/receive may return before the operation completes. For example, if the machine has a separate communication processor, a non-blocking send could simply notify the communication processor that it should begin composing and sending the message. MPI provides nonblocking sends in each of the four modes and a nonblocking receive. It also provides various utility functions for determining the completion status of a non-blocking operation.

3. **Inter-communicators.** Recollect that MPI provides two types of communicators: intra-communicators and inter-communicators. Inter-

communicators can be used for point-to-point communications between processes belonging to distinct intra-communicators.

There are many other functions available to users of MPI. If we haven't discussed a facility you need, please consult the *Standard* [4] to determine whether it is part of MPI.

## 6.2 Implementations of MPI

If you don't have an implementation of MPI, there are three versions that are freely available by anonymous ftp from the following sites.

- Argonne National Lab/Mississippi State University. The address is info.mcs.anl.gov, and the directory is pub/mpi.
- Edinburgh University. The address is ftp.epcc.ed.ac.uk, and the directory is pub/chimp/release.
- Ohio Supercomputer Center. The address is tbag.osc.edu, and the directory is pub/lam.

All of these run on networks of UNIX workstations. The Argonne/Mississippi State and Edinburgh versions also run on various parallel processors. Check the "README" files to see if your machine(s) are supported.

## 6.3 More Information on MPI

There is an MPI FAQ available by anonymous ftp at

- Mississippi State University. The address is ftp.erc.msstate.edu, and the file is pub/mpi/faq.

There are also numerous web pages devoted to MPI. A few of these are

- http://www.epm.ornl.gov/~walker/mpi. The Oak Ridge National Lab MPI web page.
- http://www.erc.msstate.edu/mpi. The Mississippi State MPI web page.
- http://www.mcs.anl.gov/mpi. The Argonne MPI web page.

Each of these sites contains a wealth of information about MPI. Of particular note, the Mississippi State page contains a bibliography of papers on MPI, and the Argonne page contains a collection of test MPI programs.

The *MPI Standard* [4] is currently available from each of the sites above. This is, of course, the definitive statement of what MPI is. So if you're not clear on something, this is the final arbiter. It also contains a large number of nice examples of uses of the various MPI functions. So it is considerably more than just a reference. Currently, several members of the MPI Forum are working on an annotated version of the MPI standard [5].

The book [2] is a tutorial introduction to MPI. It provides numerous complete examples of MPI programs.

The book [6] contains a tutorial introduction to MPI (on which this guide is based). It also contains a more general introduction to parallel processing and the programming of message-passing machines.

The Usenet newsgroup, comp.parallel.mpi, provides information on updates to all of these documents and software.

## 6.4 The Future of MPI

As it is currently defined, MPI fails to specify two critical concepts: I/O and the creation/destruction of processes. Work has already been started on the development of both I/O facilities and dynamic process creation. Information on the former can be obtained from http://lovelace.nas.nasa.gov/MPI-IO/mpi-io.html, and information on the latter can be found on the Argonne MPI web page. Significant developments are invariably posted to comp.parallel.mpi.