9 Parallel Programming with MPI

William Gropp and Ewing Lusk

Parallel computation on a Beowulf is accomplished by dividing a computation into parts and making use of multiple processes, each executing on a separate processor, to carry out these parts. Sometimes an ordinary program can be used by all the processes, but with distinct input files or parameters. In such a situation, no communication occurs among the separate tasks. When the power of a parallel computer is needed to attack a large problem with a more complex structure, however, such communication is necessary.

One of the most straightforward approaches to communication is to have the processes coordinate their activities by sending and receiving messages, much as a group of people might cooperate to perform a complex task. This approach to achieving parallelism is called *message passing*.

In this chapter and the next, we show how to write parallel programs using MPI, the Message Passing Interface. MPI is a message-passing library specification. All three parts of this description are significant.

- MPI addresses the message-passing model of parallel computation, in which processes with separate address spaces synchronize with one another and move data from the address space of one process to that of another by sending and receiving messages.[1]
- MPI specifies a library interface, that is, a collection of subroutines and their arguments. It is not a language; rather, MPI routines are called from programs written in conventional languages such as Fortran, C, and C++.
- MPI is a specification, not a particular implementation. The specification was created by the MPI Forum, a group of parallel computer vendors, computer scientists, and users who came together to cooperatively work out a community standard. The first phase of meetings resulted in a release of the standard in 1994 that is sometimes referred to as MPI-1. Once the standard was implemented and in wide use a second series of meetings resulted in a set of extensions, referred to as MPI-2. MPI refers to both MPI-1 and MPI-2.

As a specification, MPI is defined by a standards document, the way C, Fortran, or POSIX are defined. The MPI standards documents are available at `www.mpi-forum.org` and may be freely downloaded. The MPI-1 and MPI-2 standards are also available as journal issues [10, 11] and in annotated form as books

[1] Processes may be single threaded, with one program counter, or multithreaded, with multiple program counters. MPI is for communication among processes rather than threads. Signal handlers can be thought of as executing in a separate thread.

in this series [14, 4]. Implementations of MPI are available for almost all parallel computers, from clusters to the largest and most powerful parallel computers in the world. In Section 9.8 we provide a summary of the most popular cluster implementations.

A goal of the MPI Forum was to create a powerful, flexible library that could be implemented efficiently on the largest computers and provide a tool to attack the most difficult problems in parallel computing. It does not always do the simplest things in the simplest way but comes into its own as more complex functionality is needed. In this chapter and the next we work through a set of examples, starting with the simplest.

9.1 Hello World in MPI

To see what an MPI program looks like, we start with the classic "hello world" program. MPI specifies only the library calls to be used in a C, Fortran, or C++ program; consequently, all of the capabilities of the language are available. The simplest "Hello World" program is shown in Figure 9.1.

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello World\n" );
    MPI_Finalize();
    return 0;
}
```

Figure 9.1
Simple "Hello World" program in MPI.

All MPI programs must contain one call to `MPI_Init` and one to `MPI_Finalize`. All other[2] MPI routines must be called after `MPI_Init` and before `MPI_Finalize`. All C and C++ programs must also include the file '`mpi.h`'; Fortran programs must either `use` the `MPI` module or include `mpif.h`.

The simple program in Figure 9.1 is not very interesting. In particular, all processes print the same text. A more interesting version has each process identify

[2]There are a few exceptions, including `MPI_Initialized`.

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

int main( int argc, char *argv[] )
{
    int rank, size;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Hello World from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Figure 9.2
A more interesting version of "Hello World".

itself. This version, shown in Figure 9.2, illustrates several important points. Of particular note are the variables `rank` and `size`. Because MPI programs are made up of communicating processes, each process has its own set of variables. In this case, each process has its own address space containing its own variables `rank` and `size` (and `argc`, `argv`, etc.). The routine `MPI_Comm_size` returns the number of processes in the MPI job in the second argument. Each of the MPI processes is identified by a number, called the *rank*, ranging from zero to the value of `size` minus one. The routine `MPI_Comm_rank` returns in the second argument the rank of the process. The output of this program might look something like the following:

```
Hello World from process 0 of 4
Hello World from process 2 of 4
Hello World from process 3 of 4
Hello World from process 1 of 4
```

Note that the output is not ordered from processes 0 to 3. MPI does not specify the behavior of other routines or language statements such as `printf`; in particular, it does not specify the order of output from print statements.

9.1.1 Compiling and Running MPI Programs

The MPI standard does not specify how to compile and link programs (neither do C or Fortran). However, most MPI implementations provide tools to compile and

link programs.

The MPICH implementation of MPI provides instructions on setting up a makefile or project file for use with Microsoft Visual Studio. In version 1.2.1, these are

Include Path: '`<MPICH home>\include`'

Switches: `/MTd` for the debug version and `/MT` for the release version

Libraries: '`mpich.lib`' (contains MPI-1, all PMPI routines, and MPI-IO); '`mpe.lib`' contains the profiling interface to MPI-1

Library Path: '`<MPICH home>\lib`'

Include path: '`<MPICH home>\include`'

Running an MPI program (in most implementations) also requires a special program, particularly when parallel programs are started by a batch system as described in Chapter 13. Many implementations provide a program `mpirun` that can be used to start MPI programs. For example, the command

```
mpirun -np 4 helloworld
```

runs the program `helloworld` using four processes. Most MPI implementations will attempt to run each process on a different processor; most MPI implementations provide a way to select particular processors for each MPI process. In MPICH, `mpirun` should be run from within a console process.

The name and command-line arguments of the program that starts MPI programs were not specified by the original MPI standard, just as the C standard does not specify how to start C programs. However, the MPI Forum did recommend, as part of the MPI-2 standard, an `mpiexec` command and standard command-line arguments to be used in starting MPI programs. By 2002, most MPI implementations should provide `mpiexec`. This name was selected because no MPI implementation was using it (many are using `mpirun`, but with incompatible arguments). The syntax is almost the same as for the MPICH version of `mpirun`; instead of using `-np` to specify the number of processes, the switch `-n` is used:

```
mpiexec -n 4 helloworld
```

The MPI standard defines additional switches for `mpiexec`; for more details, see Section 4.1, "Portable MPI Process Startup", in the MPI-2 standard.

9.1.2 Adding Communication to Hello World

The code in Figure 9.2 does not guarantee that the output will be printed in any particular order. To force a particular order for the output, and to illustrate how data is communicated between processes, we add communication to the "Hello World" program. The revised program implements the following algorithm:

```
Find the name of the processor that is running the process
If the process has rank > 0, then
    send the name of the processor to the process with rank 0
Else
    print the name of this processor
    for each rank,
        receive the name of the processor and print it
Endif
```

This program is shown in Figure 9.3. The new MPI calls are to `MPI_Send` and `MPI_Recv` and to `MPI_Get_processor_name`. The latter is a convenient way to get the name of the processor on which a process is running. `MPI_Send` and `MPI_Recv` can be understood by stepping back and considering the two requirements that must be satisfied to communicate data between two processes:

1. Describe the data to be sent or the location in which to receive the data
2. Describe the destination (for a send) or the source (for a receive) of the data.

In addition, MPI provides a way to tag messages and to discover information about the size and source of the message. We will discuss each of these in turn.

Describing the Data Buffer. A data buffer typically is described by an address and a length, such as "`a,100`," where `a` is a pointer to 100 bytes of data. For example, the Unix `write` call describes the data to be written with an address and length (along with a file descriptor). MPI generalizes this to provide two additional capabilities: describing noncontiguous regions of data and describing data so that it can be communicated between processors with different data representations. To do this, MPI uses three values to describe a data buffer: the address, the (MPI) datatype, and the number or *count* of the items of that datatype. For example, a buffer containing four C `int`s is described by the triple "`a, 4, MPI_INT`." There are predefined MPI datatypes for all of the basic datatypes defined in C, Fortran, and C++. The most common datatypes are shown in Table 9.1.

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

int main( int argc, char *argv[] )
{
    int  numprocs, myrank, namelen, i;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    char greeting[MPI_MAX_PROCESSOR_NAME + 80];
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Get_processor_name( processor_name, &namelen );

    sprintf( greeting, "Hello, world, from process %d of %d on %s",
             myrank, numprocs, processor_name );

    if ( myrank == 0 ) {
        printf( "%s\n", greeting );
        for ( i = 1; i < numprocs; i++ ) {
            MPI_Recv( greeting, sizeof( greeting ), MPI_CHAR,
                      i, 1, MPI_COMM_WORLD, &status );
            printf( "%s\n", greeting );
        }
    }
    else {
        MPI_Send( greeting, strlen( greeting ) + 1, MPI_CHAR,
                  0, 1, MPI_COMM_WORLD );
    }

    MPI_Finalize( );
    return( 0 );
}
```

Figure 9.3
A more complex "Hello World" program in MPI. Only process 0 writes to stdout; each process sends a message to process 0.

C		Fortran	
	MPI type		MPI type
int	MPI_INT	INTEGER	MPI_INTEGER
double	MPI_DOUBLE	DOUBLE PRECISION	MPI_DOUBLE_PRECISION
float	MPI_FLOAT	REAL	MPI_REAL
long	MPI_LONG		
char	MPI_CHAR	CHARACTER	MPI_CHARACTER
		LOGICAL	MPI_LOGICAL
—	MPI_BYTE	—	MPI_BYTE

Table 9.1
The most common MPI datatypes. C and Fortran types on the same row are often but not always the same type. The type `MPI_BYTE` is used for raw data bytes and does not coorespond to any particular datatype. The C++ MPI datatypes have the same name as the C datatype, but without the `MPI_` prefix, for example, `MPI::INT`.

Describing the Destination or Source. The destination or source is specified by using the rank of the process. MPI generalizes the notion of destination and source rank by making the rank relative to a group of processes. This group may be a subset of the original group of processes. Allowing subsets of processes and using relative ranks make it easier to use MPI to write component-oriented software (more on this in Section 10.4). The MPI object that defines a group of processes (and a special communication context that will be discussed in Section 10.4) is called a *communicator*. Thus, sources and destinations are given by two parameters: a rank and a communicator. The communicator `MPI_COMM_WORLD` is predefined and contains all of the processes started by `mpirun` or `mpiexec`. As a source, the special value `MPI_ANY_SOURCE` may be used to indicate that the message may be received from any rank of the MPI processes in this MPI program.

Selecting among Messages. The "extra" argument for `MPI_Send` is a nonnegative integer *tag* value. This tag allows a program to send one extra number with the data. `MPI_Recv` can use this value either to select which message to receive (by specifying a specific tag value) or to use the tag to convey extra data (by specifying the *wild card* value `MPI_ANY_TAG`). In the latter case, the tag value of the received message is stored in the `status` argument (this is the last parameter to `MPI_Recv` in the C binding). This is a structure in C, an integer array in Fortran, and a class in C++. The tag and rank of the sending process can be accessed by referring to the appropriate element of `status` as shown in Table 9.2.

C	Fortran	C++
`status.MPI_SOURCE`	`status(MPI_SOURCE)`	`status.Get_source()`
`status.MPI_TAG`	`status(MPI_TAG)`	`status.Get_tag()`

Table 9.2
Accessing the source and tag after an `MPI_Recv`.

Determining the Amount of Data Received. The amount of data received can be found by using the routine `MPI_Get_count`. For example,

```
MPI_Get_count( &status, MPI_CHAR, &num_chars );
```

returns in `num_chars` the number of characters sent. The second argument should be the same MPI datatype that was used to receive the message. (Since many applications do not need this information, the use of a routine allows the implementation to avoid computing `num_chars` unless the user needs the value.)

Our example provides a maximum-sized buffer in the receive. It is also possible to find the amount of memory needed to receive a message by using `MPI_Probe`, as shown in Figure 9.4.

```
char *greeting;
int num_chars, src;
MPI_Status status;
...
MPI_Probe( MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status );
MPI_Get_count( &status, MPI_CHAR, &num_chars );
greeting = (char *)malloc( num_chars );
src      = status.MPI_SOURCE;
MPI_Recv( greeting, num_chars, MPI_CHAR,
          src, 1, MPI_COMM_WORLD, &status );
```

Figure 9.4
Using `MPI_Probe` to find the size of a message before receiving it.

MPI guarantees that messages are ordered and that an `MPI_Recv` after an `MPI_Probe` will receive the message that the probe returned information on as long as the same message selection criteria (source rank, communicator, and message tag) are used. Note that in this example, the source for the `MPI_Recv` is specified as `status.MPI_SOURCE`, not `MPI_ANY_SOURCE`, to ensure that the message received is the same as the one about which `MPI_Probe` returned information.

9.2 Manager/Worker Example

We now begin a series of examples illustrating approaches to parallel computations that accomplish useful work. While each parallel application is unique, a number of paradigms have emerged as widely applicable, and many parallel algorithms are variations on these patterns.

One of the most universal is the "manager/worker" or "task parallelism" approach. The idea is that the work that needs to be done can be divided by a "manager" into separate pieces and the pieces can be assigned to individual "worker" processes. Thus the manager executes a different algorithm from that of the workers, but all of the workers execute the same algorithm. Most implementations of MPI (including MPICH) allow MPI processes to be running different programs (executable files), but it is often convenient (and in some cases required) to combine the manager and worker code into a single program with the structure shown in Figure 9.5.

```
#include "mpi.h"

int main( int argc, char *argv[] )
{
    int numprocs, myrank;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

    if ( myrank == 0 )          /* manager process */
        manager_code ( numprocs );
    else                        /* worker process */
        worker_code ( );
    MPI_Finalize( );
    return 0;
}
```

Figure 9.5
Framework of the matrix-vector multiply program.

Sometimes the work can be evenly divided into exactly as many pieces as there are workers, but a more flexible approach is to have the manager keep a pool of units of work larger than the number of workers, and assign new work dynamically

to workers as they complete their tasks and send their results back to the manager. This approach, called *self-scheduling*, works well in the presence of tasks of varying sizes and/or workers of varying speeds.

We illustrate this technique with a parallel program to multiply a matrix by a vector. (A Fortran version of this same program can be found in [6].) This program is not a particularly good way to carry out this operation, but it illustrates the approach and is simple enough to be shown in its entirety. The program multiplies a square matrix `a` by a vector `b` and stores the result in `c`. The units of work are the individual dot products of the rows of `a` with the vector `b`. Thus the manager, code for which is shown in Figure 9.6, starts by initializing `a`. The manager then sends out initial units of work, one row to each worker. We use the MPI tag on each such message to encode the row number we are sending. Since row numbers start at 0 but we wish to reserve 0 as a tag with the special meaning of "no more work to do", we set the tag to one greater than the row number. When a worker sends back a dot product, we store it in the appropriate place in `c` and send that worker another row to work on. Once all the rows have been assigned, workers completing a task are sent a "no more work" message, indicated by a message with tag 0.

The code for the worker part of the program is shown in Figure 9.7. A worker initializes `b`, receives a row of `a` in a message, computes the dot product of that row and the vector `b`, and then returns the answer to the manager, again using the tag to identify the row. A worker repeats this until it receives the "no more work" message, identified by its tag of 0.

This program requires at least two processes to run: one manager and one worker. Unfortunately, adding more workers is unlikely to make the job go faster. We can analyze the cost of computation and communication mathematically and see what happens as we increase the number of workers. Increasing the number of workers will decrease the amount of computation done by each worker, and since they work in parallel, this should decrease total elapsed time. On the other hand, more workers mean more communication, and the cost of communicating a number is usually much greater than the cost of an arithmetical operation on it. The study of how the total time for a parallel algorithm is affected by changes in the number of processes, the problem size, and the speed of the processor and communication network is called *scalability analysis*. We analyze the matrix-vector program as a simple example.

First, let us compute the number of floating-point operations. For a matrix of size n, we have to compute n dot products, each of which requires n multiplications and $n-1$ additions. Thus the number of floating-point operations is $n \times (n + (n-1)) = n \times (2n-1) = 2n^2 - n$. If T_{calc} is the time it takes a processor to do one floating-point

```
#define SIZE 1000
#define MIN( x, y ) ((x) < (y) ? x : y)

void manager_code( int numprocs )
{
    double a[SIZE][SIZE], c[SIZE];

    int i, j, sender, row, numsent = 0;
    double dotp;
    MPI_Status status;

    /* (arbitrary) initialization of a */
    for (i = 0; i < SIZE; i++ )
        for ( j = 0; j < SIZE; j++ )
            a[i][j] = ( double ) j;

    for ( i = 1; i < MIN( numprocs, SIZE ); i++ ) {
        MPI_Send( a[i-1], SIZE, MPI_DOUBLE, i, i, MPI_COMM_WORLD );
        numsent++;
    }
    /* receive dot products back from workers */
    for ( i = 0; i < SIZE; i++ ) {
        MPI_Recv( &dotp, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &status );
        sender = status.MPI_SOURCE;
        row    = status.MPI_TAG - 1;
        c[row] = dotp;
        /* send another row back to this worker if there is one */
        if ( numsent < SIZE ) {
            MPI_Send( a[numsent], SIZE, MPI_DOUBLE, sender,
                      numsent + 1, MPI_COMM_WORLD );
            numsent++;
        }
        else                      /* no more work */
            MPI_Send( MPI_BOTTOM, 0, MPI_DOUBLE, sender, 0,
                      MPI_COMM_WORLD );
    }
}
```

Figure 9.6
The matrix-vector multiply program, manager code.

```
void worker_code( void )
{
    double b[SIZE], c[SIZE];
    int i, row, myrank;
    double dotp;
    MPI_Status status;

    for ( i = 0; i < SIZE; i++ ) /* (arbitrary) b initialization */
        b[i] = 1.0;

    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if ( myrank <= SIZE ) {
        MPI_Recv( c, SIZE, MPI_DOUBLE, 0, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &status );
        while ( status.MPI_TAG > 0 ) {
            row = status.MPI_TAG - 1;
            dotp = 0.0;
            for ( i = 0; i < SIZE; i++ )
                dotp += c[i] * b[i];
            MPI_Send( &dotp, 1, MPI_DOUBLE, 0, row + 1,
                      MPI_COMM_WORLD );
            MPI_Recv( c, SIZE, MPI_DOUBLE, 0, MPI_ANY_TAG,
                      MPI_COMM_WORLD, &status );
        }
    }
}
```

Figure 9.7
The matrix-vector multiply program, worker code.

operation, then the total computation time is $(2n^2 - n) \times T_{calc}$. Next, we compute the number of communications, defined as sending one floating-point number. (We ignore for this simple analysis the effect of message lengths.) Leaving aside the cost of communicating `b` (perhaps it is computed locally in a preceding step), we have to send each row of `a` and receive back one dot product answer. So the number of floating-point numbers communicated is $(n \times n) + n = n^2 + n$. If T_{comm} is the time to communicate one number, we get $(n^2 + n) \times T_{comm}$ for the total communication time. Thus the ratio of communication time to computation time is

$$\left(\frac{n^2 + n}{2n^2 - n}\right) \times \left(\frac{T_{comm}}{T_{calc}}\right).$$

In many computations the ratio of communication to computation can be reduced almost to 0 by making the problem size larger. Our analysis shows that this is not the case here. As n gets larger, the term on the left approaches $\frac{1}{2}$. Thus we can expect communication costs to prevent this algorithm from showing good speedups, even on large problem sizes.

The situation is better in the case of matrix-*matrix* multiplication, which could be carried out by a similar algorithm. We would replace the vectors `b` and `c` by matrices, send the entire matrix `b` to the workers at the beginning of the computation, and then hand out the rows of `a` as work units, just as before. The workers would compute an entire row of the product, consisting of the dot products of the row of `a` with all of the column of `b`, and then return a row of `c` to the manager.

Let us now do the scalability analysis for the matrix-matrix multiplication. Again we ignore the initial communication of `b`. The number of operations for one dot product is $n + (n + 1)$ as before, and the total number of dot products calculated is n^2. Thus the total number of operations is $n^2 \times (2n - 1) = 2n^3 - n^2$. The number of numbers communicated has gone up to $(n \times n) + (n \times n) = 2n^2$. So the ratio of communication time to computation time has become

$$\left(\frac{2n^2}{2n^3 - n^2}\right) \times \left(\frac{T_{comm}}{T_{calc}}\right),$$

which does tend to 0 as n gets larger. Thus, for large matrices the communication costs play less of a role.

Two other difficulties with this algorithm might occur as we increase the size of the problem and the number of workers. The first is that as messages get longer, the workers waste more time waiting for the next row to arrive. A solution to this problem is to "double buffer" the distribution of work, having the manager send two rows to each worker to begin with, so that a worker always has some work to do while waiting for the next row to arrive.

Another difficulty for larger numbers of processes can be that the manager can become overloaded so that it cannot assign work in a timely manner. This problem can most easily be addressed by increasing the size of the work unit, but in some cases it is necessary to parallelize the manager task itself, with multiple managers handling subpools of work units.

A more subtle problem has to do with *fairness*: ensuring that all worker processes are fairly serviced by the manager. MPI provides several ways to ensure fairness; see [6, Section 7.1.4].

9.3 Two-Dimensional Jacobi Example with One-Dimensional Decomposition

A common use of parallel computers in scientific computation is to approximate the solution of a partial differential equation (PDE). One of the most common PDEs, at least in textbooks, is the Poisson equation (here shown in two dimensions):

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ in } \Gamma \tag{9.3.1}$$

$$u = g(x,y) \text{ on } \partial\Gamma \tag{9.3.2}$$

This equation is used to describe many physical phenomena, including fluid flow and electrostatics. The equation has two parts: a differential equation applied everywhere within a domain Γ (9.3.1) and a specification of the value of the unknown u along the boundary of Γ (the notation $\partial\Gamma$ means "the boundary of Γ"). For example, if this equation is used to model the equilibrium distribution of temperature inside a region, the boundary condition $g(x,y)$ specifies the applied temperature along the boundary, $f(x,y)$ is zero, and $u(x,y)$ is the temperature within the region. To simplify the rest of this example, we will consider only a simple domain Γ consisting of a square (see Figure 9.8).

To compute an approximation to $u(x,y)$, we must first reduce the problem to finite size. We cannot determine the value of u everywhere; instead, we will approximate u at a finite number of points (x_i, y_j) in the domain, where $x_i = i \times h$ and $y_j = j \times h$. (Of course, we can define a value for u at other points in the domain by interpolating from these values that we determine, but the approximation is defined by the value of u at the points (x_i, y_j).) These points are shown as black disks in Figure 9.8. Because of this regular spacing, the points are said to make up a *regular mesh*. At each of these points, we approximate the partial derivatives with finite differences. For example,

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) \approx \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2}.$$

If we now let $u_{i,j}$ stand for our approximation to solution of Equation 9.3.1 at the point (x_i, y_j), we have the following set of simultaneous linear equations for the values of u:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = f(x_i, y_j). \tag{9.3.3}$$

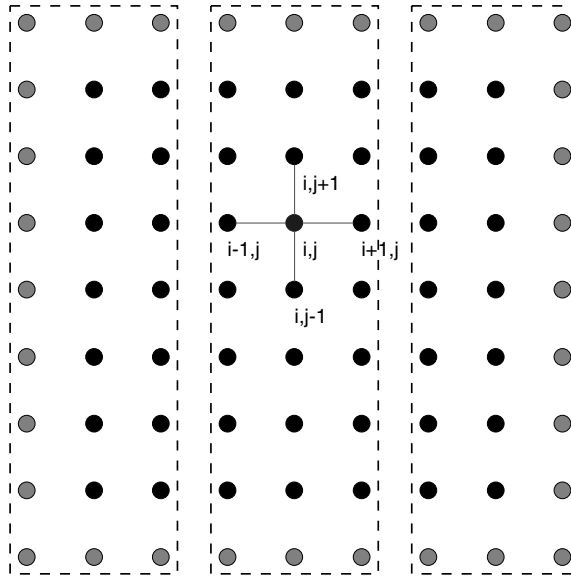

Figure 9.8
Domain and 9×9 computational mesh for approximating the solution to the Poisson problem.

For values of u along the boundary (e.g., at $x = 0$ or $y = 1$), the value of the boundary condition g is used. If $h = 1/(n+1)$ (so there are $n \times n$ points in the interior of the mesh), this gives us n^2 simultaneous linear equations to solve.

Many methods can be used to solve these equations. In fact, if you have this particular problem, you should use one of the numerical libraries described in Table 10.1. In this section, we describe a very simple (and inefficient) algorithm because, from a parallel computing perspective, it illustrates how to program more effective and general methods. The method that we use is called the *Jacobi* method for solving systems of linear equations. The Jacobi method computes successive approximations to the solution of Equation 9.3.3 by rewriting the equation as follows:

$$\begin{aligned} u_{i+1,j} \quad - \quad & 2u_{i,j} + u_{i-1,j} + u_{i,j+1} - 2u_{i,j} + u_{i,j-1} = h^2 f(x_i, y_j) \\ u_{i,j} \quad = \quad & \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2 f_{i,j}). \end{aligned} \tag{9.3.4}$$

Each step in the Jacobi iteration computes a new approximation to $u_{i,j}^{N+1}$ in terms of the surrounding values of u^N:

$$u_{i,j}^{N+1} = \frac{1}{4}(u_{i+1,j}^{N} + u_{i-1,j}^{N} + u_{i,j+1}^{N} + u_{i,j-1}^{N} - h^2 f_{i,j}). \tag{9.3.5}$$

This is our algorithm for computing the approximation to the solution of the Poisson problem. We emphasize that the Jacobi method is a poor numerical method but that the same communication patterns apply to many finite difference, volume, or element discretizations solved by iterative techniques.

In the uniprocessor version of this algorithm, the solution u is represented by a two-dimensional array `u[max_n][max_n]`, and the iteration is written as follows:

```
double u[NX+2][NY+2], u_new[NX+2][NY+2], f[NX+2][NY+2];
int    i, j;
...
for (i=1;i<=NX;i++)
   for (j=1;j<=NY;j++)
      u_new[i][j] = 0.25 * (u[i+1][j] + u[i-1][j] +
                            u[i][j+1] + u[i][j-1] - h*h*f[i][j]);
```

Here, we let `u[0][j]`, `u[n+1][j]`, `u[i][0]`, and `u[i][n+1]` hold the values of the boundary conditions g (these correspond to $u(0, y)$, $u(1, y)$, $u(x, 0)$, and $u(x, 1)$ in Equation 9.3.1). To parallelize this method, we must first decide how to decompose the data structure `u` and `u_new` across the processes. Many possible decompositions exist. One of the simplest is to divide the domain into strips as shown in Figure 9.8.

Let the local representation of the array `u` be `ulocal`; that is, each process declares an array `ulocal` that contains the part of `u` held by that process. No process has all of `u`; the data structure representing `u` is *decomposed* among all of the processes. The code that is used on each process to implement the Jacobi method is

```
for (i=i_start;i<=i_end;i++)
  for (j=1;j<=NY;j++)
    ulocal_new[i-i_start][j] =
       0.25 * (ulocal[i-i_start+1][j] + ulocal[i-i_start-1][j] +
               ulocal[i-i_start][j+1] + ulocal[i-i_start][j-1] -
               h*h*flocal[i-i_start][j]);
```

where `i_start` and `i_end` describe the strip on this process (in practice, the loop would be from zero to `i_end-i_start`; we use this formulation to maintain the correspondence with the uniprocessor code). We have defined `ulocal` so that `ulocal[0][j]` corresponds to `u[i_start][j]` in the uniprocessor version of this code. Using variable names such as `ulocal` that make it obvious which variables are part of a distributed data structure is often a good idea.

From this code, we can see what data we need to communicate. For `i=i_start` we need the values of `u[i_start-1][j]`, and for `i=i_end` we need `u[i_end+1][j]`. These values belong to the adjacent processes and must be communicated. In addition, we need a location in which to store these values. We could use a separate array, but for regular meshes the most common approach is to use *ghost* or *halo* cells, where extra space is set aside in the `ulocal` array to hold the values from neighboring processes. In this case, we need only a single column of neighboring data, so we will let `u_local[1][j]` correspond to `u[i_start][j]`. This changes the code for a single iteration of the loop to

```
exchange_nbrs( ulocal, i_start, i_end, left, right );
for (i_local=1; i_local<=i_end-i_start+1; i_local++)
  for (j=1; j<=NY; j++)
    ulocal_new[i_local][j] =
       0.25 * (ulocal[i_local+1][j] + ulocal[i_local-1][j] +
               ulocal[i_local][j+1] + ulocal[i_local][j-1] -
               h*h*flocal[i_local][j]);
```

where we have converted the `i` index to be relative to the start of `ulocal` rather than `u`. All that is left is to describe the routine `exchange_nbrs` that exchanges data between the neighboring processes. A very simple routine is shown in Figure 9.9.

We note that ISO/ANSI C (unlike Fortran) does not allow runtime dimensioning of multidimensional arrays. To keep these examples simple in C, we use compile-time dimensioning of the arrays. An alternative in C is to pass the arrays a one-dimensional arrays and compute the appropriate offsets.

The values `left` and `right` are used for the ranks of the left and right neighbors, respectively. These can be computed simply by using the following:

```
int rank, size, left, right;
...
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
left  = rank - 1;
right = rank + 1;
if (left < 0)      left  = MPI_PROC_NULL;
if (right >= size) right = MPI_PROC_NULL;
```

The special rank `MPI_PROC_NULL` indicates the edges of the mesh. If `MPI_PROC_NULL` is used as the source or destination rank in an MPI communication call, the

```
void exchange_nbrs( double ulocal[][NY+2], int i_start, int i_end,
                    int left, int right )
{
    MPI_Status status;
    int c;

    /* Send and receive from the left neighbor */
    MPI_Send( &ulocal[1][1], NY, MPI_DOUBLE, left, 0,
              MPI_COMM_WORLD );
    MPI_Recv( &ulocal[0][1], NY, MPI_DOUBLE, left, 0,
              MPI_COMM_WORLD, &status );

    /* Send and receive from the right neighbor */
    c = i_end - i_start + 1;
    MPI_Send( &ulocal[c][1], NY, MPI_DOUBLE, right, 0,
              MPI_COMM_WORLD );
    MPI_Recv( &ulocal[c+1][1], NY, MPI_DOUBLE, right, 0,
              MPI_COMM_WORLD, &status );
}
```

Figure 9.9
A simple version of the neighbor exchange code. See the text for a discussion of the limitations of this routine.

operation is ignored. MPI also provides routines to compute the neighbors in a regular mesh of arbitrary dimension and to help an application choose a decomposition that is efficient for the parallel computer.

The code in `exchange_nbrs` will work with most MPI implementations for small values of `n` but, as described in Section 10.3, is not good practice (and will fail for values of `NY` greater than an implementation-defined threshold). A better approach in MPI is to use the `MPI_Sendrecv` routine when exchanging data between two processes, as shown in Figure 9.10.

In Sections 10.3 and 10.7, we discuss other implementations of the exchange routine that can provide higher performance. MPI support for more scalable decompositions of the data is described in Section 10.3.2.

9.4 Collective Operations

A *collective* operation is an MPI function that is called by all processes belonging to a communicator. (If the communicator is `MPI_COMM_WORLD`, this means all

```
/* Better exchange code.  */
void exchange_nbrs( double ulocal[][NY+2], int i_start, int i_end,
                    int left, int right )
{
    MPI_Status status;
    int c;

    /* Send and receive from the left neighbor */
    MPI_Sendrecv( &ulocal[1][1], NY, MPI_DOUBLE, left, 0,
                  &ulocal[0][1], NY, MPI_DOUBLE, left, 0,
                  MPI_COMM_WORLD, &status );

    /* Send and receive from the right neighbor */
    c = i_end - i_start + 1;
    MPI_Sendrecv( &ulocal[c][1], NY, MPI_DOUBLE, right, 0,
                  &ulocal[c+1][1], NY, MPI_DOUBLE, right, 0,
                  MPI_COMM_WORLD, &status );
}
```

Figure 9.10
A better version of the neighbor exchange code.

processes, but MPI allows collective operations on other sets of processes as well.) Collective operations involve communication and also sometimes computation, but since they describe particular patterns of communication and computation, the MPI implementation may be able to optimize them beyond what is possible by expressing them in terms of MPI point-to-point operations such as `MPI_Send` and `MPI_Recv`. The patterns are also easier to express with collective operations.

Here we introduce two of the most commonly used collective operations and show how the communication in a parallel program can be expressed entirely in terms of collective operations with no individual `MPI_Send`s or `MPI_Recv`s at all. The program shown in Figure 9.11 computes the value of π by numerical integration. Since

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x)|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4},$$

we can compute π by integrating the function $f(x) = 4/(1 + x^2)$ from 0 to 1. We compute an approximation by dividing the interval [0,1] into some number of subintervals and then computing the total area of these rectangles by having each process compute the areas of some subset. We could do this with a manager/worker

algorithm, but here we preassign the work. In fact, each worker can compute its set of tasks, and so the "manager" can be a worker, too, instead of just managing the pool of work. The more rectangles there are, the more work there is to do and the more accurate the resulting approximation of π is. To experiment, let us make the number of subintervals a command-line argument. (Although the MPI standard does not guarantee that any process receives command-line arguments, in most implementations, especially for Beowulf clusters, one can assume that at least the process with rank 0 can use `argc` and `argv`, although they may not be meaningful until after `MPI_Init` is called.) In our example, process 0 sets n, the number of subintervals, to `argv[1]`. Once a process knows n, it can claim approximately $\frac{1}{n}$ of the work by claiming every nth rectangle, starting with the one numbered by its own rank. Thus process j computes the areas of rectangles j , $j+n$, $j+2n$, and so on.

Not all MPI implementations make the command-line arguments available to *all* processes, however, so we start by having process 0 send n to each of the other processes. We could have a simple loop, sending n to each of the other processes one at a time, but this is inefficient. If we know that the same message is to be delivered to all the other processes, we can ask the MPI implementation to do this in a more efficient way than with a series of `MPI_Send`s and `MPI_Recv`s.

Broadcast (`MPI_Bcast`) is an example of an MPI *collective* operation. A collective operation must be called by all processes in a communicator. This allows an implementation to arrange the communication and computation specified by a collective operation in a special way. In the case of `MPI_Bcast`, an implementation is likely to use a tree of communication, sometimes called a spanning tree, in which process 0 sends its message to a second process, then both processes send to two more, and so forth. In this way most communication takes place in parallel, and all the messages have been delivered in $\log_2 n$ steps.

The precise semantics of `MPI_Bcast` is sometimes confusing. The first three arguments specify a message with (address, count, datatype) as usual. The fourth argument (called the *root* of the broadcast) specifies which of the processes owns the data that is being sent to the other processes. In our case it is process 0. `MPI_Bcast` acts like an `MPI_Send` on the root process and like an `MPI_Recv` on all the other processes, but the call itself looks the same on each process. The last argument is the communicator that the collective call is *over*. All processes in the communicator must make this same call. Before the call, n is valid only at the root; after `MPI_Bcast` has returned, all processes have a copy of the value of n.

Next, each process, including process 0, adds up the areas of its rectangles into the local variable `mypi`. Instead of sending these values to one process and having

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f(double a) { return (4.0 / (1.0 + a*a)); }

int main(int argc,char *argv[])
{
  int  n, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264 3;
  double mypi, pi, h, sum, x;
  double startwtime = 0.0, endwtime;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  if (myid == 0) {
      startwtime = MPI_Wtime();
      n = atoi(argv[1]);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += f(x);
  }
  mypi = h * sum;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (myid == 0) {
      endwtime = MPI_Wtime();
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
      printf("wall clock time = %f\n", endwtime-startwtime);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 9.11
Computing π using collective operations.

that process add them up, however, we use another collective operation, `MPI_Reduce`. `MPI_Reduce` performs not only collective communication but also collective computation. In the call

```
MPI_Reduce( &mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);
```

the sixth argument is again the root. All processes call `MPI_Reduce`, and the root process gets back a result in the second argument. The result comes from performing an arithmetic operation, in this case summation (specified by the fifth argument), on the data items on all processes specified by the first, third, and fourth arguments.

Process 0 concludes by printing out the answer, the difference between this approximation and a previously computed accurate value of π, and the time it took to compute it. This illustrates the use of `MPI_Wtime`.

`MPI_Wtime` returns a double-precision floating-point number of seconds. This value has no meaning in itself, but the *difference* between two such values is the wall-clock time between the two calls. Note that calls on two different processes are not guaranteed to have any relationship to one another, unless the MPI implementation promises that the clocks on different processes are synchronized (see `MPI_WTIME_IS_GLOBAL` in any of the MPI books).

The routine `MPI_Allreduce` computes the same result as `MPI_Reduce` but returns the result to all processes, not just the root process. For example, in the Jacobi iteration, it is common to use the two-norm of the difference between two successive iterations as a measure of the convergence of the solution.

```
...
norm2local = 0.0;
for (ii=1; ii<i_end-i_start+1; ii++)
    for (jj=1; jj<NY; jj++)
        norm2local += ulocal[ii][jj] * ulocal[ii][jj];
MPI_Allreduce( &norm2local, &norm2, 1, MPI_DOUBLE,
               MPI_COMM_WORLD, MPI_SUM );
norm2 = sqrt( norm2 );
```

Note that `MPI_Allreduce` is not a routine for computing the norm of a vector. It merely combines values contributed from each process in the communicator.

9.5 Parallel Monte Carlo Computation

One of the types of computation that is easiest to parallelize is the *Monte Carlo* family of algorithms. In such computations, a random number generator is used to create a number of independent trials. Statistics done with the outcomes of the trials provide a solution to the problem.

We illustrate this technique with another computation of the value of π. If we select points at random in the unit square $[0,1] \times [0,1]$ and compute the percentage of them that lies inside the quarter circle of radius 1, then we will be approximating $\frac{\pi}{4}$. (See [6] for a more detailed discussion together with an approach that does not use a parallel random number generator.) We use the SPRNG parallel random number generator (`sprng.cs.fsu.edu`). The code is shown in Figure 9.12.

The defaults in SPRNG make it extremely easy to use. Calls to the `sprng` function return a random number between 0.0 and 1.0, and the stream of random numbers on the different processes is independent. We control the *grain size* of the parallelism by the constant `BATCHSIZE`, which determines how much computation is done before the processes communicate. Here a million points are generated, tested, and counted before we collect the results to print them. We use `MPI_Bcast` to distribute the command-line argument specifying the number of batches, and we use `MPI_Reduce` to collect at the end of each batch the number of points that fell inside the quarter circle, so that we can print the increasingly accurate approximations to π.

9.6 Installing MPICH under Windows 2000

The MPICH implementation of MPI [5] is one of the most popular versions of MPI. Thanks to support from Microsoft, an open-source version is available for Windows NT and Windows 2000. This implementation supports TCP/IP, VIA, and shared-memory communication. This release is available at `www.mcs.anl.gov/mpi/mpich/mpich-nt`, which also contains complete installation instructions for the current version.

This implementation includes

- all source code for MPICH,
- simple example programs like the ones in this chapter,
- performance benchmarking programs,
- the MPE profiling library, and
- the Jumpshot performance visualization system.

```
#include "mpi.h"
#include <stdio.h>
#define SIMPLE_SPRNG            /* simple interface  */
#define USE_MPI                 /* use MPI           */
#include "sprng.h"              /* SPRNG header file */
#define BATCHSIZE 1000000

int main( int argc, char *argv[] )
{
    int i, j, numin = 0, totalin, total, numbatches, rank, numprocs;
    double x, y, approx, pi = 3.14159265358979323846264 3;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if ( rank == 0 ) {
        numbatches = atoi( argv[1] );
    }
    MPI_Bcast( &numbatches, 1, MPI_INT, 0, MPI_COMM_WORLD );
    for ( i = 0; i < numbatches; i++ ) {
        for ( j = 0; j < BATCHSIZE; j++ ) {
            x = sprng( ); y = sprng( );
            if ( x * x + y * y  < 1.0 )
                numin++;
        }
        MPI_Reduce( &numin, &totalin, 1, MPI_INT, MPI_SUM, 0,
                    MPI_COMM_WORLD );
        if ( rank == 0 ) {
            total = BATCHSIZE * ( i + 1 ) * numprocs;
            approx = 4.0 * ( (double) totalin / total );
            printf( "pi = %.16f; error = %.16f, points = %d\n",
                    approx, pi - approx, total );
        }
    }
    MPI_Finalize( );
}
```

Figure 9.12
Computing π using the Monte Carlo method.

The distribution is available in several forms. The one appropriate for most users is '`mpich.nt.1.2.1.zip`', which contains everything you need to compile and run MPI programs.

The various distributions are as follows:

Full source tree: '`mpich.nt.1.2.1.src.exe`' is a self-extracting WinZip archive.

Binary distribution: '`mpich.nt.1.2.1.zip`' contains the runtime DLLs, services to start MPI programs, the Visual C++ software developers kit (SDK), and the gcc SDK. An SDK is needed to compile applications but not to run them. You should use `setup.exe` to install the DLLs and at least one job launcher on all machines that will be used to run MPI processes.

Software Developers Kit: '`mpich.nt.1.2.1.zip`' contains the SDK, which includes libraries and `include` files to compile an MPI application using Microsoft Visual C++ and/or Visual Fortran 6.

gcc Software Developers Kit: '`mpich.nt.1.2.1.zip`' contains the gcc SDK, which includes libraries and `include` files to compile an MPI application using `gcc` and the Cygnus tools.

Using MPIRun. The easiest way to run an MPI program is with

```
MPIRun -np 2 myapp.exe
```

The `MPIRun` program accepts several other arguments; see the documentation for a complete list. The most commonly used are as follows:

`-env name=value` to pass environment variables to the program. A typical use of this is to pass environment variables that MPICH itself uses. For example,

```
MPIRun -np 2 -env "MPICH_USE_POLLING=1|MPICH_SINGLETHREAD=1" mpptest.exe
```

is appropriate for getting the lowest latency in message passing. See the documentation under "Subtle Configuration Options" for more details on the environment variables that affect MPICH.

`-localonly n` to run n processes on the local machine, using shared memory to communicate between processes. This is often helpful when debugging an MPI program.

The most flexible way to run a program is with a configuration file. The command-line `mpirun file.cfg` uses the specified configuration file ('`file.cfg`') to run the parallel program. This format allows MPMD (Multiple Program Multiple Data) programs.

Two tools help in running MPICH programs:

- `MPIConfig` attempts to find the machines in your cluster and saves their names in the registry. This tool provides a graphical user interface for finding and modifying the machines available for MPI programs. Using `MPIConfig` allows `MPIRun` to accept the `-np` argument to select the number of processes.
- `MPIRegister` allows you to provide your account name and password only once when running MPI programs. Without this, `MPIRun` will prompt for a username and password.

Here is a configuration file that starts a manager process ('`manager.exe`') and five worker processes:

```
exe d:\Projects\Me\worker.exe
hosts
roadrunner 1 d:\projects\Me\manager.exe
roadrunner 5
```

9.7 Tools

A number of tools are available for developing, testing, and tuning MPI programs. Although they are distributed with MPICH, they can be used with other MPI implementations as well.

9.7.1 Profiling Libraries

The MPI Forum decided not to standardize any particular tool but rather to provide a general mechanism for intercepting calls to MPI functions, which is the sort of capability that tools need. The MPI standard requires that any MPI implementation provide two entry points for each MPI function: its normal `MPI_` name and a corresponding `PMPI` version. This strategy allows a user to write a custom version of `MPI_Send`, for example, that carries out whatever extra functions might be desired, calling `PMPI_Send` to perform the usual operations of `MPI_Send`. When the user's custom versions of MPI functions are placed in a library and the library precedes the usual MPI library in the link path, the user's custom code will be invoked around all MPI functions that have been replaced.

MPICH provides three such "profiling libraries" and some tools for creating more. These libraries are easily used by passing an extra argument to MPICH's `mpicc` command for compiling and linking.

`-mpilog` causes a file to be written containing timestamped events. The log file can be examined with tools such as Jumpshot (see below).

`-mpitrace` causes a trace of MPI calls, tagged with process rank in `MPI_COMM_WORLD` to be written to `stdout`.

`-mpianim` shows a simple animation of message traffic while the program is running.

The profiling libraries are part of the MPE subsystem of MPICH, which is separately distributable and works with any MPI implementation.

9.7.2 Visualizing Parallel Program Behavior

The detailed behavior of a parallel program is surprisingly difficult to predict. It is often useful to examine a graphical display that shows the exact sequence of states that each process went through and what messages were exchanged at what times and in what order. The data for such a tool can be collected by means of a profiling library. One tool for looking at such log files is Jumpshot [17]. A screenshot of Jumpshot in action is shown in Figure 9.13.

The horizontal axis represents time, and there is a horizontal line for each process. The states that processes are in during a particular time interval are represented by colored rectangles. Messages are represented by arrows. It is possible to zoom in for microsecond-level resolution in time.

9.8 MPI Implementations for Clusters

Many implementations of MPI are available for clusters; Table 9.3 lists some of the available implementations. These range from commercially supported software to supported, freely available software to distributed research project software.

9.9 MPI Routine Summary

This section provide a quick summary of the MPI routines used in this chapter for C, Fortran, and C++. Although these are only a small fraction of the routines available in MPI, they are sufficient for many applications.

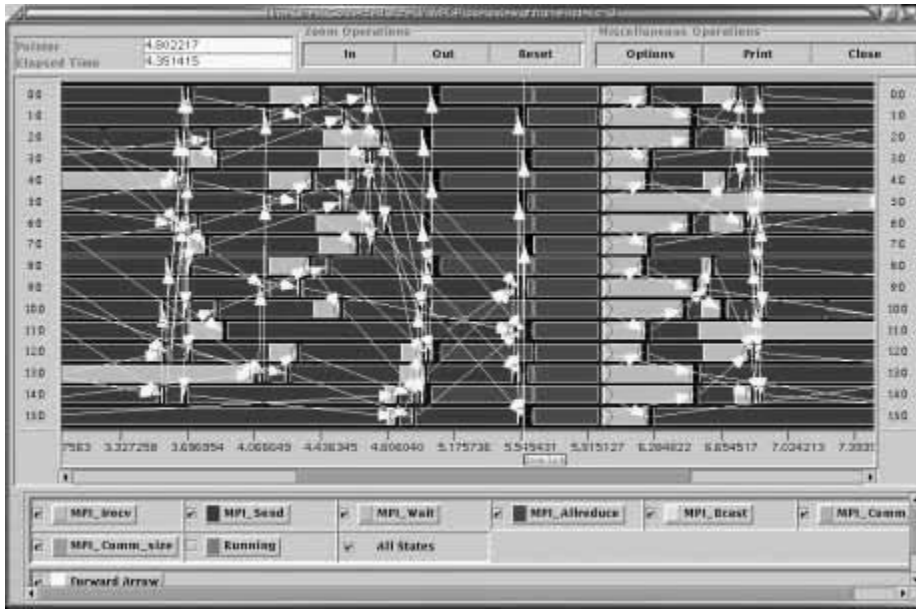

Figure 9.13
Jumpshot displaying message traffic

Name	URL
MPICH	`www.mcs.anl.gov/mpi/mpich`
MPI-FM	`www-csag.ucsd.edu/projects/comm/mpi-fm.html`
MPI/Pro	`www.mpi-softtech.com`
MP-MPICH	`www.lfbs.rwth-aachen.de/users/joachim/MP-MPICH/`
WMPI	`www.criticalsoftware.com`

Table 9.3
Some MPI implementations for Windows.

C Routines.

int **MPI_Init**(int *argc, char ***argv)

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

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

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

int **MPI_Reduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int **MPI_Finalize**()

double **MPI_Wtime**()

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

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

int **MPI_Probe**(int source, int tag, MPI_Comm comm, MPI_Status *status)

int **MPI_Sendrecv**(void *sendbuf, int sendcount,MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)

int **MPI_Allreduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Fortran routines.

MPI_INIT(ierror)
integer ierror

MPI_COMM_SIZE(comm, size, ierror)
integer comm, size, ierror

MPI_COMM_RANK(comm, rank, ierror)
integer comm, rank, ierror

MPI_BCAST(buffer, count, datatype, root, comm, ierror)
<type> buffer(*)
integer count, datatype, root, comm, ierror

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
<type> sendbuf(*), recvbuf(*)
integer count, datatype, op, root, comm, ierror

MPI_FINALIZE(ierror)
integer ierror

double precision **MPI_WTIME**()

MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, ierror

MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
<type> buf(*)
integer count, datatype, source, tag, comm,
status(MPI_STATUS_SIZE), ierror

MPI_PROBE(source, tag, comm, status, ierror)
logical flag
integer source, tag, comm, status(MPI_STATUS_SIZE), ierror

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,recvcount,
recvtype, source, recvtag, comm, status, ierror)
<type> sendbuf(*), recvbuf(*)
integer sendcount, sendtype, dest, sendtag, recvcount, recvtype,
source, recvtag, comm, status(MPI_STATUS_SIZE), ierror

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierror)
<type> sendbuf(*), recvbuf(*)
integer count, datatype, op, comm, ierror

C++ routines.
void MPI::Init(int& argc, char**& argv)

void MPI::Init()

int MPI::Comm::Get_rank() const

int MPI::Comm::Get_size() const

void MPI::Intracomm::Bcast(void* buffer, int count, const Datatype& datatype,
int root) const

void MPI::Intracomm::Reduce(const void* sendbuf, void* recvbuf, int count,
const Datatype& datatype, const Op& op, int root) const

void MPI::Finalize()

double MPI::Wtime()

int MPI::Status::Get_source() const

int MPI::Status::Get_tag() const

void MPI::Comm::Recv(void* buf, int count, const Datatype& datatype,
int source, int tag, Status& status) const

void MPI::Comm::Recv(void* buf, int count, const Datatype& datatype,
int source, int tag) const

void MPI::Comm::Send(const void* buf, int count, const Datatype& datatype,
int dest, int tag) const

void MPI::Comm::Probe(int source,int tag, Status& status) const

void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount,
const Datatype& sendtype, int dest, int sendtag, void *recvbuf,
int recvcount, const Datatype& recvtype, int source, int recvtag,
Status& status) const

void MPI::Intracomm::Allreduce(const void* sendbuf, void* recvbuf, int count,
const Datatype& datatype, const Op& op) const

This page intentionally left blank

10 Advanced Topics in MPI Programming

William Gropp and Ewing Lusk

In this chapter we continue our exploration of parallel programming with MPI. We describe capabilities that are more specific to MPI rather than part of the message-passing programming model in general. We cover the more advanced features of MPI sometimes called MPI-2, such as dynamic process management, parallel I/O, and remote memory access.

10.1 Dynamic Process Management in MPI

A new aspect of the MPI-2 standard is the ability of an MPI program to create new MPI processes and communicate with them. (In the original MPI specification, the number of processes was fixed at startup.) MPI calls this capability (together with related capabilities such as connecting two independently started MPI jobs) *dynamic process management*. Three main issues are introduced by this collection of features:

- maintaining simplicity and flexibility;
- interacting with the operating system, a parallel process manager, and perhaps a job scheduler; and
- avoiding race conditions that could compromise correctness.

The key to avoiding race conditions is to make creation of new processes a collective operation, over both the processes creating the new processes and the new processes being created.

10.1.1 Intercommunicators

Recall that an MPI communicator consists of a group of processes together with a communication context. Strictly speaking, the communicators we have dealt with so far are *intracommunicators*. There is another kind of communicator, called an *intercommunicator*. An intercommunicator binds together a communication context and *two* groups of processes, called (from the point of view of a particular process) the *local* group and the *remote* group. Processes are identified by rank in group, but ranks in an intercommunicator always refer to the processes in the remote group. That is, an `MPI_Send` using an intercommunicator sends a message to the process with the destination rank in the *remote* group of the intercommunicator. Collective operations are also defined for intercommunicators; see [7, Chapter 7] for details.

10.1.2 Spawning New MPI Processes

We are now in a position to explain exactly how new MPI processes are created by an already running MPI program. The MPI function that does this is `MPI_Comm_spawn`. Its key features are the following.

- It is collective over the communicator of processes initiating the operation (called the *parents*) and also collective with the calls to `MPI_Init` in the processes being created (called the *children*). That is, the `MPI_Comm_spawn` does not return in the parents until it has been called in all the parents and `MPI_Init` has been called in all the children.
- It returns an intercommunicator in which the local group contains the parents and the remote group contains the children.
- The new processes, which must call `MPI_Init`, have their own `MPI_COMM_WORLD`, consisting of all the processes created by this one collective call to `MPI_Comm_spawn`.
- The function `MPI_Comm_get_parent`, called by the children, returns an intercommunicator with the children in the local group and the parents in the remote group.
- The collective function `MPI_Intercomm_merge` may be called by parents and children to create a normal (intra)communicator containing all the processes, both old and new, but for many communication patterns this is not necessary.

10.1.3 Revisiting Matrix-Vector Multiplication

Here we illustrate the use of `MPI_Comm_spawn` by redoing the matrix-vector multiply program of Section 9.2. Instead of starting with a fixed number of processes, we compile separate executables for the manager and worker programs, start the manager with

```
mpiexec -n 1 manager <number-of-workers>
```

and then let the manager create the worker processes dynamically. The program for the manager is shown in Figure 10.1, and the code for the workers is shown in Figure 10.2. Here we assume that only the manager has the matrix `a` and the vector `b` and broadcasts them to the workers after the workers have been created.

Let us consider in detail the call in the manager that creates the worker processes.

```
MPI_Spawn( "worker", MPI_ARGV_NULL, numworkers, MPI_INFO_NULL,
           0, MPI_COMM_SELF, &workercomm, MPI_ERRCODES_IGNORE );
```

It has eight arguments. The first is the name of the executable to be run by the new processes. The second is the null-terminated argument vector to be passed to

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 10000

int main( int argc, char *argv[] )
{
    double a[SIZE][SIZE], b[SIZE], c[SIZE];
    int i, j, row, numworkers;
    MPI_Status status;
    MPI_Comm workercomm;

    MPI_Init( &argc, &argv );
    if ( argc != 2 || !isnumeric( argv[1] ))
        printf( "usage: %s <number of workers>\n", argv[0] );
    else
        numworkers = atoi( argv[1] );

    MPI_Spawn( "worker", MPI_ARGV_NULL, numworkers, MPI_INFO_NULL,
               0, MPI_COMM_SELF, &workercomm, MPI_ERRCODES_IGNORE );
    ...
    /* initialize a and b */
    ...
    /* send b to each worker */
    MPI_Bcast( b, SIZE, MPI_DOUBLE, MPI_ROOT, workercomm );
    ...
    /* then normal manager code as before*/
    ...
    MPI_Finalize();
    return 0;
}
```

Figure 10.1
Dynamic process matrix-vector multiply program, manager part.

all of the new processes; here we are passing no arguments at all, so we specify the special value `MPI_ARGV_NULL`. Next is the number of new processes to create. The fourth argument is an MPI "Info" object, which can be used to specify special environment- and/or implementation-dependent parameters, such as the names of the nodes to start the new processes on. In our case we leave this decision to the MPI implementation or local process manager, and we pass the special value

MPI_INFO_NULL. The next argument is the "root" process for this call to MPI_-Comm_spawn; it specifies which process in the communicator given in the following argument is supplying the valid arguments for this call. The communicator we are using consists here of just the one manager process, so we pass MPI_COMM_SELF. Next is the address of the new intercommunicator to be filled in, and finally an array of error codes for examining possible problems in starting the new processes. Here we use MPI_ERRCODES_IGNORE to indicate that we will not be looking at these error codes.

Code for the worker processes that are spawned is shown in Figure 10.2. It is essentially the same as the worker subroutine in the preceding chapter but is an MPI program in itself. Note the use of intercommunicator broadcast in order to receive the vector b from the parents. We free the parent intercommunicator with MPI_Comm_free before exiting.

10.1.4 More on Dynamic Process Management

For more complex examples of the use of MPI_Comm_spawn, including how to start processes with different executables or different argument lists, see [7, Chapter 7]. MPI_Comm_spawn is only the most basic of the functions provided in MPI for dealing with a dynamic MPI environment. By querying the attribute MPI_UNIVERSE_SIZE, you can find out how many processes can be usefully created. Separately started MPI computations can find each other and connect with MPI_Comm_connect and MPI_Comm_accept. Processes can exploit non-MPI connections to "bootstrap" MPI communication. These features are explained in detail in [7].

10.2 Fault Tolerance

Communicators are a fundamental concept in MPI. Their sizes are fixed at the time they are created, and the efficiency and correctness of collective operations rely on this fact. Users sometimes conclude from the fixed size of communicators that MPI provides no mechanism for writing fault-tolerant programs. Now that we have introduced intercommunicators, however, we are in a position to discuss how this topic might be addressed and how you might write a manager-worker program with MPI in such a way that it would be fault tolerant. In this context we mean that if one of the worker processes terminates abnormally, instead of terminating the job you will be able to carry on the computation with fewer workers, or perhaps dynamically replace the lost worker.

The key idea is to create a separate (inter)communicator for each worker and

```
#include "mpi.h"

int main( int argc, char *argv[] )
{
    int numprocs, myrank;
    double b[SIZE], c[SIZE];
    int i, row, myrank;
    double dotp;
    MPI_Status status;
    MPI_Comm parentcomm;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

    MPI_Comm_get_Parentp &parentcomm );

    MPI_Bcast( b, SIZE, MPI_DOUBLE, 0, parentcomm );

    ...
    /* same as worker code from original matrix-vector multiply */
    ...

    MPI_Comm_free(parentcomm );
    MPI_Finalize( );
    return 0;
}
```

Figure 10.2
Dynamic process matrix-vector multiply program, worker part.

use it for communications with that worker rather than use a communicator that contains all of the workers. If an implementation returns "invalid communicator" from an `MPI_Send` or `MPI_Recv` call, then the manager has lost contact only with one worker and can still communicate with the other workers through the other, still-intact communicators. Since the manager will be using separate communicators rather than separate ranks in a larger communicator to send and receive message from the workers, it might be convenient to maintain an array of communicators and a parallel array to remember which row has been last sent to a worker, so that if that worker disappears, the same row can be assigned to a different worker.

Figure 10.3 shows these arrays and how they might be used. What we are doing

```
/* highly incomplete */

MPI_Comm worker_comms[MAX_WORKERS];
int last_row_sent[MAX_WORKERS];

rc = MPI_Send( a[numsent], SIZE, MPI_DOUBLE, 0, numsent+1,
               worker_comms[sender] );
if ( rc != MPI_SUCCESS ) {
    /* Check that error class is one we can recover from */
    ...
    MPI_Comm_spawn( "worker" , ... );
```

Figure 10.3
Fault-tolerant manager.

with this approach is recognizing that two-party communication can be made fault tolerant, since one party can recognize the failure of the other and take appropriate action. A normal MPI communicator is not a two-party system and cannot be made fault tolerant without changing the semantics of MPI communication. If, however, the communication in an MPI program can be expressed in terms of intercommunicators, which are inherently two-party (the local group and the remote group), then fault tolerance can be achieved.

Note that while the MPI standard, through the use of intercommunicators, makes it possible to write an implementation of MPI that encourages fault-tolerant programming, the MPI standard itself does not require MPI implementations to continue past an error. This is a "quality of implementation" issue and allows the MPI implementor to trade performance for the ability to continue after a fault. As this section makes clear, however, there is nothing in the MPI standard that stands in the way of fault tolerance, and the two primary MPI implementations for Beowulf clusters, MPICH and LAM/MPI, both endeavor to support some style of fault tolerance for applications.

10.3 Revisiting Mesh Exchanges

The discussion of the mesh exchanges for the Jacobi problem in Section 9.3 concentrated on the algorithm and data structures, particularly the ghost-cell exchange. In this section, we return to that example and cover two other important issues: the

use of blocking and nonblocking communications and communicating noncontiguous data.

10.3.1 Blocking and Nonblocking Communication

Consider the following simple code (note that this is similar to the simple version of `exchange_nbrs` in Section 9.3):

```
if (rank == 0) {
    MPI_Send( sbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD );
    MPI_Recv( rbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
}
else if (rank == 1) {
    MPI_Send( sbuf, n, MPI_INT, 0, 0, MPI_COMM_WORLD );
    MPI_Recv( rbuf, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
}
```

What happens with this code? It looks like process 0 is sending a message to process 1 and that process 1 is sending a message to process 0. But more is going on here. Consider the steps that the MPI implementation must take to make this code work:

1. Copy the data from the `MPI_Send` into a temporary, system-managed buffer.

2. Once the `MPI_Send` completes (on each process), start the `MPI_Recv`. The data that was previously copied into a system buffer by the `MPI_Send` operation can now be delivered into the user's buffer (`rbuf` in this case).

This approach presents two problems, both related to the fact that data must be copied into a system buffer to allow the `MPI_Send` to complete. The first problem is obvious: any data motion takes time and reduces the performance of the code. The second problem is more subtle and important: the amount of available system buffer space always has a limit. For values of `n` in the above example that exceed the available buffer space, the above code will *hang*: neither `MPI_Send` will complete, and the code will wait forever for the other process to start an `MPI_Recv`. This is true for *any* message-passing system, not just MPI. The amount of buffer space available for buffering a message varies among MPI implementations, ranging from many megabytes to as little as 128 bytes.

How can we write code that sends data among several processes and that does not rely on the availability of system buffers? One approach is to carefully order the send and receive operations so that each send is guaranteed to have a matching

receive. For example, we can swap the order of the MPI_Send and MPI_Recv in the code for process 1:

```
if (rank == 0) {
    MPI_Send( sbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD );
    MPI_Recv( rbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
}
else if (rank == 1) {
    MPI_Recv( rbuf, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( sbuf, n, MPI_INT, 0, 0, MPI_COMM_WORLD );
}
```

However, this can be awkward to implement, particularly for more complex communication patterns; in addition, it does not address the extra copy that may be performed by MPI_Send.

The approach used by MPI, following earlier message-passing systems as well as nonblocking sockets (see [6, Chapter 9]), is to split the send and receive operations into two steps: one to initiate the operation and one to complete the operation. Other operations, including other communication operations, can be issued between the two steps. For example, an MPI receive operation can be initiated by a call to MPI_Irecv and completed with a call to MPI_Wait. Because the routines that initiate these operations do not wait for them to complete, they are called *nonblocking* operations. The "I" in the routine name stands for "immediate"; this indicates that the routine may return immediately without completing the operation. The arguments to MPI_Irecv are the same as for MPI_Recv except for the last (status) argument. This is replaced by an MPI_Request value; it is a *handle* that is used to identify an initiated operation. To complete a nonblocking operation, the request is given to MPI_Wait, along with a status argument; the status argument serves the same purpose as status for an MPI_Recv. Similarly, the nonblocking counterpart to MPI_Send is MPI_Isend; this has the same arguments as MPI_Send with the addition of an MPI_Request as the last argument (in C). Using these routines, our example becomes the following:

```
if (rank == 0) {
    MPI_Request req1, req2;
    MPI_Isend( sbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &req1 );
    MPI_Irecv( rbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &req2 );
    MPI_Wait( &req1, &status );
    MPI_Wait( &req2, &status );
```

```
}
else if (rank == 1) {
    MPI_Request req1, req2;
    MPI_Irecv( rbuf, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &req1 );
    MPI_Isend( sbuf, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &req2 );
    MPI_Wait( &req1, &status );
    MPI_Wait( &req2, &status );
}
```

The buffer `sbuf` provided to `MPI_Isend` must not be modified until the operation is completed with `MPI_Wait`. Similarly, the buffer `rbuf` provided to `MPI_Irecv` must not be modified or read until the `MPI_Irecv` is completed.

The nonblocking communication routines allow the MPI implementation to wait until the message can be sent directly from one user buffer to another (e.g., from `sbuf` to `rbuf`) without requiring any copy or using any system buffer space.

Because it is common to start multiple nonblocking operations, MPI provides routines to test or wait for completion of any one, all, or some of the requests. For example, `MPI_Waitall` waits for all requests in an array of requests to complete. Figure 10.4 shows the use of nonblocking communication routines for the Jacobi example.[1]

MPI nonblocking operations are not the same as asynchronous operations. The MPI standard does not require that the data transfers overlap computation with communication. MPI specifies only the semantics of the operations, not the details of the implementation choices. The MPI nonblocking routines are provided primarily for correctness (avoiding the limitations of system buffers) and performance (avoidance of copies).

10.3.2 Communicating Noncontiguous Data in MPI

The one-dimensional decomposition used in the Jacobi example (Section 9.3) is simple but does not scale well and can lead to performance problems. We can analyze the performance of the Jacobi following the discussion in Section 9.2. Let the time to communicate n bytes be

$$T_{comm} = s + rn,$$

where s is the *latency* and r is the (additional) time to communicate one byte. The time to compute one step of the Jacobi method, using the one-dimensional decomposition in Section 9.3, is

[1] On many systems, calling `MPI_Isend` before `MPI_Irecv` will improve performance.

```
void exchange_nbrs( double ulocal[][NY+2], int i_start, int i_end,
                    int left, int right )
{
    MPI_Status  statuses[4];
    MPI_Request requests[4];
    int c;

    /* Begin send and receive from the left neighbor */
    MPI_Isend( &ulocal[1][1], NY, MPI_DOUBLE, left, 0,
               MPI_COMM_WORLD, &requests[0] );
    MPI_Irecv( &ulocal[0][1], NY, MPI_DOUBLE, left, 0,
               MPI_COMM_WORLD, &requests[1] );

    /* Begin send and receive from the right neighbor */
    c = i_end - i_start + 1;
    MPI_Isend( &ulocal[c][1], NY, MPI_DOUBLE, right, 0,
               MPI_COMM_WORLD, &requests[2] );
    MPI_Irecv( &ulocal[c+1][1], NY, MPI_DOUBLE, right, 0,
               MPI_COMM_WORLD, &requests[3] );

    /* Wait for all communications to complete */
    MPI_Waitall( 4, requests, statuses );
}
```

Figure 10.4
Nonblocking exchange code for the Jacobi example.

$$\frac{5n}{p}f + 2(s + rn),$$

where f is the time to perform a floating-point operation and p is the number of processes. Note that the cost of communication is independent of the number of processes; eventually, this cost will dominate the calculation. Hence, a better approach is to use a two-dimensional decomposition, as shown in Figure 10.5.

The time for one step of the Jacobi method with a two-dimensional decomposition is just

$$\frac{5n}{p}f + 4\left(s + r\frac{n}{\sqrt{p}}\right).$$

This is faster than the one-dimensional decomposition as long as

$$n > \frac{2}{1 - 4/\sqrt{p}}\frac{s}{r}$$

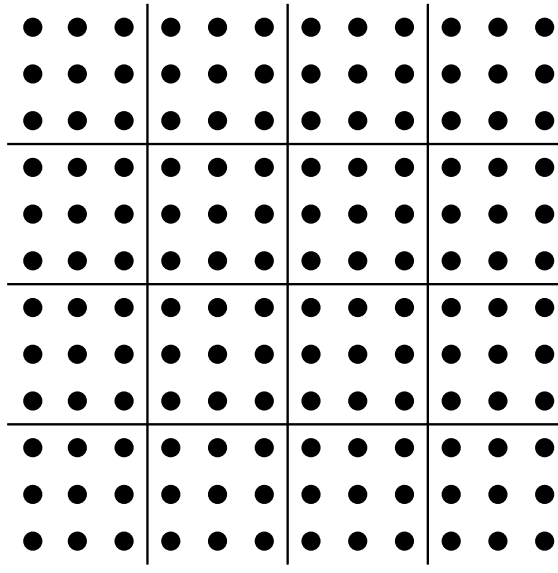

Figure 10.5
Domain and 9×9 computational mesh for approximating the solution to the Poisson problem using a two-dimensional decomposition.

(assuming $p \geq 16$). To implement this decomposition, we need to communicate data to four neighbors, as shown in Figure 10.6.

The left and right edges can be sent and received by using the same code as for the one-dimensional case. The top and bottom edges have noncontiguous data. For example, the top edge needs to send the tenth, sixteenth, and twenty-second element. There are four ways to move this data:

1. Each value can be sent separately. Because of the high latency of message passing, this approach is inefficient and normally should not be used.

2. The data can be copied into a temporary buffer using a simple loop, for example,

```
for (i=0; i<3; i++) {
    tmp[i] = u_local[i][6];
}
MPI_Send( tmp, 3, MPI_DOUBLE, ..  );
```

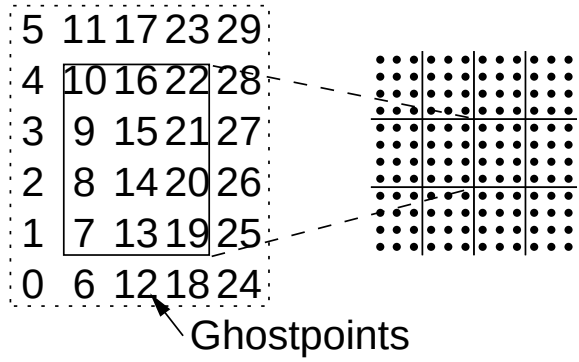

Figure 10.6
Locations of mesh points in `ulocal` for a two-dimensional decomposition.

This is a common approach and, for some systems and MPI implementations, may be the most efficient.

3. MPI provides two routines to pack and unpack a buffer. These routines are `MPI_Pack` and `MPI_Unpack`. A buffer created with these routines should be sent and received with MPI datatype `MPI_PACKED`. We note, however, that these routines are most useful for complex data layouts that change frequently within a program.

4. MPI provides a way to construct new datatypes representing any data layout. These routines can be optimized by the MPI implementation, in principle providing better performance than the user can achieve using a simple loop [16]. In addition, using these datatypes is crucial to achieving high performance with parallel I/O.

MPI provides several routines to create datatypes representing common patterns of memory. These new datatypes are called *derived* datatypes. For this case, `MPI_Type_vector` is what is needed to create a new MPI datatype representing data values separated by a constant *stride*. In this case, the stride is `NY+2`, and the number of elements is `i_end-i_start+1`.

```
MPI_Type_vector( i_end - i_start + 1, 1, NY+2,
                 MPI_DOUBLE, &vectype );
MPI_Type_commit( &vectype );
```

The second argument is a *block count* and is the number of the basic datatype items (`MPI_DOUBLE` in this case); this is useful particularly in multicomponent PDE problems. The routine `MPI_Type_commit` must be called to *commit* the MPI datatype;

this call allows the MPI implementation to optimize the datatype (the optimization is not included as part of the routines that create MPI datatypes because some complex datatypes are created recursively from other derived datatypes).

Using an MPI derived datatype representing a strided data pattern, we can write a version of `exchange_nbr` for a two-dimensional decomposition of the mesh; the code is shown in Figure 10.7. Note that we use the same derived datatype `vectype` for the sends and receives at the top and bottom by specifying the first element into which data is moved in the array `u_local` in the MPI calls.

When a derived datatype is no longer needed, it should be freed with `MPI_Type_free`. Many other routines are available for creating datatypes; for example, `MPI_Type_indexed` is useful for scatter-gather patterns, and `MPI_Type_create_struct` can be used for an arbitrary collection of memory locations.

10.4 Motivation for Communicators

Communicators in MPI serve two purposes. The most obvious purpose is to describe a collection of processes. This feature allows collective routines, such as `MPI_Bcast` or `MPI_Allreduce`, to be used with any collection of processes. This capability is particularly important for hierarchical algorithms, and also facilitates dividing a computation into subtasks, each of which has its own collection of processes. For example, in the manager-worker example in Section 9.2, it may be appropriate to divide each task among a small collection of processes, particularly if this causes the problem description to reside only in the fast memory cache. MPI communicators are perfect for this; the MPI routine `MPI_Comm_split` is the only routine needed when creating new communicators. Using ranks relative to a communicator for specifying the source and destination of messages also facilitates dividing parallel tasks among smaller but still parallel subtasks, each with its own communicator.

A more subtle but equally important purpose of the MPI communicator involves the *communication context* that each communicator contains. This context is essential for writing software libraries that can be safely and robustly combined with other code, both other libraries and user-specific application code, to build complete applications. Used properly, the communication context guarantees that messages are received by appropriate routines *even if other routines are not as careful.* Consider the example in Figure 10.8 (taken from [6, Section 6.1.2]). In this example, there are two routines, provided by separate libraries or software modules. One, `SendRight`, sends a message to the right neighbor and receives from the left. The other, `SendEnd`, sends a message from process 0 (the leftmost) to the last process

```
void exchange_nbrs2d( double ulocal[][NY+2],
                      int i_start, int i_end, int j_start, int j_end,
                      int left, int right, int top, int bottom,
                      MPI_Datatype vectype )
{
    MPI_Status  statuses[8];
    MPI_Request requests[8];
    int c;

    /* Begin send and receive from the left neighbor */
    MPI_Isend( &ulocal[1][1], NY, MPI_DOUBLE, left, 0,
               MPI_COMM_WORLD, &requests[0] );
    MPI_Irecv( &ulocal[0][1], NY, MPI_DOUBLE, left, 0,
               MPI_COMM_WORLD, &requests[1] );

    /* Begin send and receive from the right neighbor */
    c = i_end - i_start + 1;
    MPI_Isend( &ulocal[c][1], NY, MPI_DOUBLE, right, 0,
               MPI_COMM_WORLD, &requests[2] );
    MPI_Irecv( &ulocal[c+1][1], NY, MPI_DOUBLE, right, 0,
               MPI_COMM_WORLD, &requests[3] );

    /* Begin send and receive from the top neighbor */
    MPI_Isend( &ulocal[1][NY], 1, vectype, top, 0,
               MPI_COMM_WORLD, &requests[4] );
    MPI_Irecv( &ulocal[1][NY+1], 1, vectype, top, 0,
               MPI_COMM_WORLD, &requests[5] );

    /* Begin send and receive from the bottom neighbor */
    MPI_Isend( &ulocal[1][1], 1, vectype, bottom, 0,
               MPI_COMM_WORLD, &requests[6] );
    MPI_Irecv( &ulocal[1][0], 1, vectype, bottom, 0,
               MPI_COMM_WORLD, &requests[7] );

    /* Wait for all communications to complete */
    MPI_Waitall( 8, requests, statuses );
}
```

Figure 10.7
Nonblocking exchange code for the Jacobi problem for a two-dimensional decomposition of the mesh.

(the rightmost). Both of these routines use `MPI_ANY_SOURCE` instead of a particular source in the `MPI_Recv` call. As Figure 10.8 shows, the messages can be confused, causing the program to receive the wrong data. How can we prevent this situation? Several approaches will *not* work. One is to avoid the use of `MPI_ANY_SOURCE`. This fixes this example, but only if both `SendRight` and `SendEnd` follow this rule. The approach may be adequate (though fragile) for code written by a single person or team, but it isn't adequate for libraries. For example, if `SendEnd` was written by a commercial vendor and did not use `MPI_ANY_SOURCE`, but `SendRight`, written by a different vendor or an inexperienced programmer, did use `MPI_ANY_SOURCE`, then the program would still fail, and it would look like `SendEnd` was at fault (because the message from `SendEnd` was received first).

Another approach that does not work is to use message tags to separate messages. Again, this can work if one group writes all of the code and is very careful about allocating message tags to different software modules. However, using `MPI_ANY_TAG` in an MPI receive call can still bypass this approach. Further, as shown in Figure 6.5 in [6], even if `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are not used, it is still possible for separate code modules to receive the wrong message.

The communication context in an MPI communicator provides a solution to these problems. The routine `MPI_Comm_dup` creates a new communicator from an input communicator that contains the same processes (in the same rank order) but with a new communication context. MPI messages sent in one communication context can be received only in that context. Thus, any software module or library that wants to ensure that all of its messages will be seen only within that library needs only to call `MPI_Comm_dup` at the beginning to get a new communicator. All well-written libraries that use MPI create a private communicator used only within that library.

Enabling the development of libraries was one of the design goals of MPI. In that respect MPI has been very successful. Many libraries and applications now use MPI, and, because of MPI's portability, most of these run on Beowulf clusters. Table 10.1 provides a partial list of libraries that use MPI to provide parallelism. More complete descriptions and lists are available at `www.mcs.anl.gov/mpi/libraries` and at `sal.kachinatech.com/C/3`.

10.5 More on Collective Operations

One of the strengths of MPI is its collection of scalable collective communication and computation routines. Figure 10.9 shows the capabilities of some of the most important collective communication routines. As an example of their utility, we

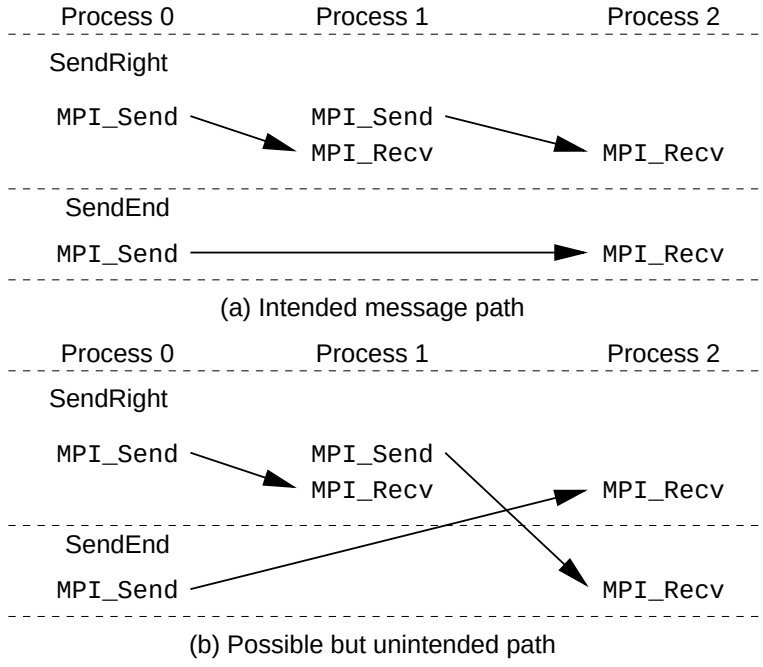

Figure 10.8
Two possible message-matching patterns when `MPI_ANY_SOURCE` is used in the `MPI_Recv` calls (from [6]).

consider a simple example.

Suppose we want to gather the names of all of the nodes that our program is running on, and we want all MPI processes to have this list of names. This is an easy task using `MPI_Allgather`:

```
char my_hostname[MAX_LEN], all_names[MAX_PROCS][MAX_LEN];
MPI_Allgather( my_hostname, MAX_LEN, MPI_CHAR,
               all_names, MAX_LEN, MPI_CHAR, MPI_COMM_WORLD );
```

This code assumes that no hostname is longer than `MAX_LEN` characters (including the trailing null). A better code would check this:

```
char my_hostname[MAX_LEN], all_names[MAX_PROCS][MAX_LEN];
MPI_Allreduce( &my_name_len, &max_name_len, 1, MPI_INT, MPI_MAX,
               MPI_COMM_WORLD );
if (max_name_len > MAX_LEN) {
```

Library	Description	URL
PETSc	Linear and nonlinear solvers for PDEs	`www.mcs.anl.gov/petsc`
Aztec	Parallel iterative solution of sparse linear systems	`www.cs.sandia.gov/CRF/aztec1.html`
Cactus	Framework for PDE solutions	`www.cactuscode.org`
FFTW	Parallel FFT	`www.fftw.org`
PPFPrint	Parallel print	`www.llnl.gov/sccd/lc/ptcprint`
HDF	Parallel I/O for Hierarchical Data Format (HDF) files	`hdf.ncsa.uiuc.edu/Parallel_HDF`
NAG	Numerical library	`www.nag.co.uk/numeric/fd/FDdescription.asp`
ScaLAPACK	Parallel linear algebra	`www.netlib.org/scalapack`
SPRNG	Scalable pseudorandom number generator	`sprng.cs.fsu.edu`

Table 10.1
A sampling of libraries that use MPI.

```
        printf( "Error: names too long (%d)", max_name_len );
    }
    MPI_Allgather( my_hostname, MAX_LEN, MPI_CHAR,
                   all_names, MAX_LEN, MPI_CHAR, MPI_COMM_WORLD );
```

Both of these approaches move more data than necessary, however. An even better approach is to first gather the size of each processor's name and then gather exactly the number of characters needed from each processor. This uses the "v" (for vector) version of the allgather routine, `MPI_Allgatherv`, as shown in Figure 10.10.

This example provides a different way to accomplish the action of the example in Section 9.3. Many parallel codes can be written with MPI collective routines instead of MPI point-to-point communication; such codes often have a simpler logical structure and can benefit from scalable implementations of the collective communications routines.

10.6 Parallel I/O

MPI-2 provides a wide variety of parallel I/O operations, more than we have space to cover here. See [7, Chapter 3] for a more thorough discussion of I/O in MPI.

Figure 10.9
Schematic representation of collective data movement in MPI.

The fundamental idea in MPI's approach to parallel I/O is that a file is opened collectively by a set of processes that are all given access to the same file. MPI thus associates a communicator with the file, allowing a flexible set of both individual and collective operations on the file.

```
mylen = strlen(my_hostname) + 1;  /* Include the trailing null */
MPI_Allgather( &mylen, 1, MPI_INT, all_lens, 1, MPI_INT,
               MPI_COMM_WORLD );
totlen = all_lens[size-1];
for (i=0; i<size-1; i++) {
    displs[i+1] = displs[i] + all_lens[i];
    totlen      += all_lens[i];
}
all_names = (char *)malloc( totlen );
if (!all_names) MPI_Abort( MPI_COMM_WORLD, 1 );
MPI_Allgatherv( my_hostname, mylen, MPI_CHAR,
                all_names, all_lens, displs, MPI_CHAR,
                MPI_COMM_WORLD );
/* Hostname for the jth process is &all_names[displs[j]] */
```

Figure 10.10
Using `MPI_Allgather` and `MPI_Allgatherv`.

10.6.1 A Simple Example

We first provide a simple example of how processes write contiguous blocks of data into the same file in parallel. Then we give a more complex example, in which the data in each process is not contiguous but can be described by an MPI datatype.

For our first example, let us suppose that after solving the Poisson equation as we did in Section 9.3, we wish to write the solution to a file. We do not need the values of the ghost cells, and in the one-dimensional decomposition the set of rows in each process makes up a contiguous area in memory, which greatly simplifies the program. The I/O part of the program is shown in Figure 10.11.

Recall that the data to be written from each process, not counting ghost cells but including the boundary data, is in the array `ulocal[i][j]` for `i=i_start` to `i_end` and `j=0` to `NY+1`.

Note that the type of an MPI file object is `MPI_File`. Such file objects are opened and closed much the way normal files are opened and closed. The most significant difference is that opening a file is a collective operation over a group of processes specified by the communicator in the first argument of `MPI_File_open`. A single process can open a file by specifying the single-process communicator `MPI_COMM_SELF`. Here we want all of the processes to share the file, and so we use `MPI_COMM_WORLD`.

In our discussion of dynamic process management, we mentioned `MPI_Info` ob-

```
MPI_File outfile;
size = NX * (NY + 2);
MPI_File_open( MPI_COMM_WORLD, "solutionfile",
               MPI_MODE_CREATE | MPI_MODE_WRONLY,
               MPI_INFO_NULL, &outfile );
MPI_File_set_view( outfile,
              rank * (NY+2) * (i_end - i_start) * sizeof(double),
              MPI_DOUBLE, MPI_DOUBLE, "native", MPI_INFO_NULL );
MPI_File_write( outfile, &ulocal[1][0], size, MPI_DOUBLE,
               MPI_STATUS_IGNORE );
MPI_File_close( &outfile );
```

Figure 10.11
Parallel I/O of Jacobi solution. Note that this choice of file view works only for a single output step; if output of multiple steps of the Jacobi method are needed, the arguments to `MPI_File_set_view` must be modified.

jects. An MPI info object is a collection of `key=value` pairs that can be used to encapsulate a variety of special-purpose information that may not be applicable to all MPI implementations. In this section we will use `MPI_INFO_NULL` whenever this type of argument is required, since we have no special information to convey. For details about `MPI_Info`, see [7, Chapter 2].

The part of the file that will be seen by each process is called the file *view* and is set for each process by a call to `MPI_File_set_view`. In our example the call is

```
MPI_File_set_view( outfile, rank * (NY+2) * ( ... ),
                   MPI_DOUBLE, MPI_DOUBLE, "native", MPI_INFO_NULL )
```

The first argument identifies the file; the second is the displacement (in bytes) into the file of where the process's view of the file is to start. Here we simply multiply the size of the data to be written by the process's rank, so that each process's view starts at the appropriate place in the file. The type of this argument is `MPI_Offset`, which can be expected to be a 64-bit integer on systems that support large files.

The next argument is called the *etype* of the view; it specifies the unit of data in the file. Here it is just `MPI_DOUBLE`, since we will be writing some number of doubles. The next argument is called the *filetype*; it is a flexible way of describing noncontiguous views in the file. In our case, with no noncontiguous units to be written, we can just use the etype, `MPI_DOUBLE`. In general, any MPI predefined or derived datatype can be used for both etypes and filetypes. We explore this use in more detail in the next example.

The next argument is a string defining the *data representation* to be used. The native representation says to represent data on disk exactly as it is in memory, which provides the fastest I/O performance, at the possible expense of portability. We specify that we have no extra information by providing `MPI_INFO_NULL` for the final argument.

The call to `MPI_File_write` is then straightforward. The data to be written is a contiguous array of doubles, even though it consists of several rows of the (distributed) matrix. On each process it starts at `&ulocal[0][1]` so the data is described in (address, count, datatype) form, just as it would be for an MPI message. We ignore the status by passing `MPI_STATUS_IGNORE`. Finally we (collectively) close the file with `MPI_File_close`.

10.6.2 A More Complex Example

Parallel I/O requires more than just calling `MPI_File_write` instead of write. The key idea is to identify the object (across processes), rather than the contribution from each process. We illustrate this with an example of a regular distributed array.

The code in Figure 10.12 writes out an array that is distributed among processes with a two-dimensional decomposition. To illustrate the expressiveness of the MPI interface, we show a complex case where, as in the Jacobi example, the distributed array is surrounded by ghost cells. This example is covered in more depth in Chapter 3 of *Using MPI 2* [7], including the simpler case of a distributed array without ghost cells.

This example may look complex, but each step is relatively simple.

1. Set up a communicator that represents a virtual array of processes that matches the way that the distributed array is distributed. This approach uses the `MPI_Cart_create` routine and uses `MPI_Cart_coords` to find the coordinates of the calling process in this array of processes. This particular choice of process ordering is important because it matches the ordering required by `MPI_Type_create_subarray`.

2. Create a *file view* that describes the part of the file that this process will write to. The MPI routine `MPI_Type_create_subarray` makes it easy to construct the MPI datatype that describes this region of the file. The arguments to this routine specify the dimensionality of the array (two in our case), the global size of the array, the local size (that is, the size of the part of the array on the calling process), the location of the local part (`start_indices`), the ordering of indices (column major is `MPI_ORDER_FORTRAN` and row major is `MPI_ORDER_C`), and the basic datatype.

```
/* no. of processes in vertical and horizontal dimensions
   of process grid */
dims[0] = 2;   dims[1] = 3;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
/* global indices of the first element of the local array */

/* no. of rows and columns in global array*/
gsizes[0] = m;    gsizes[1] = n;

lsizes[0] = m/dims[0];   /* no. of rows in local array */
lsizes[1] = n/dims[1];   /* no. of columns in local array */

start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(comm, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                  MPI_INFO_NULL);

/* create a derived datatype that describes the layout of the local
   array in the memory buffer that includes the ghost area. This is
   another subarray datatype! */
memsizes[0] = lsizes[0] + 8; /* no. of rows in allocated array */
memsizes[1] = lsizes[1] + 8; /* no. of columns in allocated array */
start_indices[0] = start_indices[1] = 4;
/* indices of the first element of the local array in the
   allocated array */
MPI_Type_create_subarray(2, memsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_FLOAT, &memtype);
MPI_Type_commit(&memtype);
MPI_File_write_all(fh, local_array, 1, memtype, &status);
MPI_File_close(&fh);
```

Figure 10.12
C program for writing a distributed array that is also noncontiguous in memory because of a ghost area (derived from an example in [7]).

3. Open the file for writing (`MPI_MODE_WRONLY`), and set the file view with the datatype we have just constructed.

4. Create a datatype that describes the data to be written. We can use `MPI_Type_create_subarray` here as well to define the part of the local array that does *not* include the ghost points. If there were no ghost points, we could instead use `MPI_FLOAT` as the datatype with a count of `lsizes[0]*lsizes[1]` in the call to `MPI_File_write_all`.

5. Perform a collective write to the file with `MPI_File_write_all`, and close the file.

By using MPI datatypes to describe both the data to be written and the destination of the data in the file with a collective file write operation, the MPI implementation can make the best use of the I/O system. The result is that file I/O operations performed with MPI I/O can achieve hundredfold improvements in performance over using individual Unix I/O operations [15].

10.7 Remote Memory Access

The message-passing programming model requires that both the sender and the receiver (or all members of a communicator in a collective operation) participate in moving data between two processes. An alternative model where one process controls the communication, called one-sided communication, can offer better performance and in some cases a simpler programming model. MPI-2 provides support for this one-sided approach. The MPI-2 model was inspired by the work on the bulk synchronous programming (BSP) model [9] and the Cray SHMEM library used on the massively parallel Cray T3D and T3E computers [1].

In one-sided communication, one process may *put* data directly into the memory of another process, without that process using an explicit receive call. For this reason, this also called *remote memory access* (RMA).

Using RMA involves four steps:

1. Describe the memory into which data may be put.
2. Allow access to the memory.
3. Begin put operations (e.g., with `MPI_Put`).
4. Complete all pending RMA operations.

The first step is to describe the region of memory into which data may be placed by an `MPI_Put` operation (also accessed by `MPI_Get` or updated by `MPI_Accumulate`). This is done with the routine `MPI_Win_create`:

```
MPI_Win win;
double ulocal[MAX_NX][NY+2];

MPI_Win_create( ulocal, (NY+2)*(i_end-i_start+3)*sizeof(double),
           sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win );
```

The input arguments are, in order, the array `ulocal`, the size of the array in bytes, the size of a basic unit of the array (`sizeof(double)` in this case), a "hint" object, and the communicator that specifies which processes may use RMA to access the array. `MPI_Win_create` is a collective call over the communicator. The output is an MPI *window object* `win`. When a window object is no longer needed, it should be freed with `MPI_Win_free`.

RMA operations take place between two sentinels. One begins a period where access is allowed to a window object, and one ends that period. These periods are called *epochs*.[2] The easiest routine to use to begin and end epochs is `MPI_Win_fence`. This routine is collective over the processes that created the window object and both ends the previous epoch and starts a new one. The routine is called a "fence" because all RMA operations before the fence complete before the fence returns, and any RMA operation initiated by another process (in the epoch begun by the matching fence on that process) does not start until the fence returns. This may seem complex, but it is easy to use. In practice, `MPI_Win_fence` is needed only to separate RMA operations into groups. This model closely follows the BSP and Cray SHMEM models, though with the added ability to work with any subset of processes.

Three routines are available for initiating the transfer of data in RMA. These are `MPI_Put`, `MPI_Get`, and `MPI_Accumulate`. All are nonblocking in the same sense MPI point-to-point communication is nonblocking (Section 10.3.1). They complete at the end of the epoch that they start in, for example, at the closing `MPI_Win_fence`. Because these routines specify both the source and destination of data, they have more arguments than do the point-to-point communication routines. The arguments can be easily understood by taking them a few at a time.

[2] MPI has two kinds of epochs for RMA: an *access epoch* and an *exposure epoch*. For the example used here, the epochs occur together, and we refer to both of them as just epochs.

1. The first three arguments describe the *origin* data; that is, the data on the calling process. These are the usual "buffer, count, datatype" arguments.

2. The next argument is the rank of the *target* process. This serves the same function as the destination of an `MPI_Send`. The rank is relative to the communicator used when creating the MPI window object.

3. The next three arguments describe the destination buffer. The `count` and `datatype` arguments have the same meaning as for an `MPI_Recv`, but the buffer location is specified as an offset from the beginning of the memory specified to `MPI_Win_create` on the target process. This offset is in units of the displacement argument of the `MPI_Win_create` and is usually the size of the basic datatype.

4. The last argument is the MPI window object.

Note that there are no MPI requests; the `MPI_Win_fence` completes all preceding RMA operations. `MPI_Win_fence` provides a collective synchronization model for RMA operations in which all processes participate. This is called *active target* synchronization.

With these routines, we can create a version of the mesh exchange that uses RMA instead of point-to-point communication. Figure 10.13 shows one possible implementation.

Another form of access requires no MPI calls (not even a fence) at the target process. This is called *passive target* synchronization. The origin process uses `MPI_Win_lock` to begin an access epoch and `MPI_Win_unlock` to end the access epoch.[3] Because of the passive nature of this type of RMA, the local memory (passed as the first argument to `MPI_Win_create`) should be allocated with `MPI_Alloc_mem` and freed with `MPI_Free_mem`. For more information on passive target RMA operations, see [7, Chapter 6]. Also note that as of 2001, few MPI implementations support passive target RMA operation. More implementations are expected to support these operations in 2002.

A more complete discussion of remote memory access can be found in [7, Chapters 5 and 6]. Note that MPI implementations are just beginning to provide the RMA routines described in this section. Most current RMA implementations emphasize functionality over performance. As implementations mature, however, the performance of RMA operations will also improve.

[3]The names `MPI_Win_lock` and `MPI_Win_unlock` are really misnomers; think of them as begin-RMA and end-RMA.

```
void exchang_nbrs( double u_local[][NY+2], int i_start, int i_end,
                   int left, int right, MPI_Win win )
{
    MPI_Aint left_ghost_disp, right_ghost_disp;
    int      c;

    MPI_Win_fence( 0, win );
    /* Put the left edge into the left neighbors rightmost
       ghost cells.  See text about right_ghost_disp */
    right_ghost_disp = 1 + (NY+2) * (i_end-i-start+2);
    MPI_Put( &u_local[1][1], NY, MPI_DOUBLE,
            left, right_ghost_disp, NY, MPI_DOUBLE, win );
    /* Put the right edge into the right neighbors leftmost ghost
       cells */
    left_ghost_disp = 1;
    c = i_end - i_start + 1;
    MPI_Put( &u_local[c][1], NY, MPI_DOUBLE,
             right, left_ghost_disp, NY, MPI_DOUBLE, win );

    MPI_Win_fence( 0, win )
}
```

Figure 10.13
Neighbor exchange using MPI remote memory access.

10.8 Using C++ and Fortran 90

MPI-1 defined bindings to C and Fortran 77. These bindings were very similar; the only major difference was the handling of the error code (returned in C, set through the last argument in Fortran 77). In MPI-2, a binding was added for C++, and an MPI module was defined for Fortran 90.

The C++ binding provides a lightweight model that is more than just a C++ version of the C binding but not a no-holds-barred object-oriented model. MPI objects are defined in the `MPI` namespace. Most MPI objects have corresponding classes, such as `Datatype` for `MPI_Datatype`. Communicators and requests are slightly different. There is an abstract base class `Comm` for general communicators with four derived classes: `Intracomm`, `Intercomm`, `Graphcomm`, and `Cartcomm`. Most communicators are `Intracomm`s; `GraphComm` and `CartComm` are derived from `Intracomm`. Requests have two derived classes: `Prequest` for persistent requests and `Grequest` for generalized requests (new in MPI-2). Most MPI operations are

methods on the appropriate objects; for example, most point-to-point and collective communications are methods on the communicator. A few routines, such as `Init` and `Finalize`, stand alone. A simple MPI program in C++ is shown in Figure 10.14.

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

int main( int argc, char *argv[] )
{
    int data;
    MPI::Init();

    if (MPI::COMM_WORLD.Get_rank() == 0) {
        // Broadcast data from process 0 to all others
        cout << "Enter an int" << endl;
        data << cin;
    }
    MPI::COMM_WORLD.Bcast( data, 1, MPI::INT, 0 );

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

Figure 10.14
Simple MPI program in C++.

The C++ binding for MPI has a few quirks. One is that the multiple completion operations such as `MPI::Waitall` are methods on requests, even though there is no unique request to use for these methods. Another is the C++ analogue to `MPI_Comm_dup`. In the C++ binding, `MPI::Comm` is an abstract base class (ABC). Since it is impossible to create an instance of an abstract base class, there can be no general "dup" function that returns a new `MPI::Comm`. Since it is possible in C++ to create a reference to an ABC, however, MPI defines the routine (available only in the C++ binding) `MPI::Clone` that returns a reference to a new communicator.

Two levels of Fortran 90 support are provided in MPI. The basic support provides an '`mpif.h`' include file. The extended support provides an `MPI` module. The module makes it easy to detect the two most common errors in Fortran MPI programs: forgetting to provide the variable for the error return value and forgetting to declare status as an array of size `MPI_STATUS_SIZE`. There are a few drawbacks.

Fortran derived datatypes cannot be directly supported (the Fortran 90 language provides no way to handle an arbitrary type). Often, you can use the first element of the Fortran 90 derived type. Array sections should not be used in receive operations, particularly nonblocking communication (see Section 10.2.2 in the MPI-2 standard for more information). Another problem is that while Fortran 90 enables the user to define MPI interfaces in the MPI module, a different Fortran 90 interface file must be used for each combination of Fortran datatype and array dimension (scalars are different from arrays of dimension one, etc.). This leads to a Fortran 90 MPI module library that is often (depending on the Fortran 90 compiler) far larger than the entire MPI library. However, particularly during program development, the MPI module is very helpful.

10.9 MPI, OpenMP, and Threads

The MPI standard was carefully written to be a thread-safe specification. That means that the design of MPI doesn't include concepts such as "last message" or "current pack buffer" that are not well defined when multiple threads are present. MPI implementations can choose whether to provide thread-safe *implementations*. Allowing this choice is particularly important because thread safety usually comes at the price of performance due to the extra overhead required to ensure that internal data structures are not modified inconsistently by two different threads. Most early MPI implementations were not thread safe.

MPI-2 introduced four levels of thread safety that an MPI implementation could provide. The lowest level, `MPI_THREAD_SINGLE`, allows only single threaded programs. The next level, `MPI_THREAD_FUNNELED`, allows multiple threads provided that all MPI calls are made in a single thread; most MPI implementations provide `MPI_THREAD_FUNNELED`. The next level, `MPI_THREAD_SERIALIZED`, allows many user threads to make MPI calls, but only one thread at a time. The highest level of support, `MPI_THREAD_MULTIPLE`, allows any thread to call any MPI routine.

Understanding the level of thread support is important when combining MPI with approaches to thread-based parallelism. OpenMP [12] is a popular and powerful language for specifying thread-based parallelism. While OpenMP provides some tools for general threaded parallelism, one of the most common uses is to parallelize a loop. If the loop contains no MPI calls, then OpenMP may be combined with MPI. For example, in the Jacobi example, OpenMP can be used to parallelize the loop computation:

```
exchange_nbrs( u_local, i_start, i_end, left, right );
```

```
#pragma omp for
for (i_local=1; i<=i_end-i_start+1; i++)
  for (j=1; j<=NY; j++)
    ulocal_new[i_local][j] =
       0.25 * (ulocal[i_local+1][j] + ulocal[i_local-1][j] +
               ulocal[i_local][j+1] + ulocal[i_local][j-1] -
               h*h*flocal[i_local][j]);
```

This exploits the fact that MPI was designed to work well with other tools, leveraging improvements in compilers and threaded parallelism.

10.10 Measuring MPI Performance

Many tools have been developed for measuring performance. The best is always your own application, but a number of tests are available that can give a more general overview of the performance of MPI on a cluster. Measuring communication performance is actually quite tricky; see [8] for a discussion of some of the issues in making reproducible measurements of performance. That paper describes the methods used in the `mpptest` program for measuring MPI performance.

10.10.1 mpptest

The `mpptest` program allows you to measure many aspects of the performance of any MPI implementation. The most common MPI performance test is the Ping-Pong test (see Section 8.2). The `mpptest` program provides Ping-Pong tests for the different MPI communication modes, as well as providing a variety of tests for collective operations and for more realistic variations on point-to-point communication, such as halo communication (like that in Section 9.3) and communication that does not reuse the same memory locations (thus benefiting from using data that is already in memory cache). The `mpptest` program can also test the performance of some MPI-2 functions, including `MPI_Put` and `MPI_Get`.

Using mpptest. The `mpptest` program is distributed with MPICH in the directory '`examples/perftest`'. You can also download it separately from `www.mcs.anl.gov/mpi/perftest`.

10.10.2 SKaMPI

The SKaMPI test suite [13] is a comprehensive test of MPI performance, covering virtually all of the MPI-1 communication functions.

One interesting feature of the SKaMPI benchmarks is the online tables showing the performance of MPI implementations on various parallel computers, ranging from Beowulf clusters to parallel vector supercomputers.

10.10.3 High Performance LINPACK

Perhaps the best known benchmark in technical computing is the LINPACK Benchmark, discussed in Section 8.3. The version of this benchmark that is appropriate for clusters is the High Performance LINPACK (HPL). Obtaining and running this benchmark is relatively easy, though getting good performance can require a significant amount of effort. In addition, as pointed out in Section 8.3, while the LINPACK benchmark is widely known, it tends to significantly overestimate the achieveable performance for many applications.

The HPL benchmark depends on another library, the basic linear algebra subroutines (BLAS), for much of the computation. Thus, to get good performance on the HPL benchmark, you must have a high-quality implementation of the BLAS. Fortunately, several sources of these routines are available. You can often get implementations of the BLAS from the CPU vendor directly, sometimes at no cost.

HPL. Download the HPL package from `www.netlib.org/benchmark/hpl`:

```
% tar zxf hpl.tgz
% cd hpl
```

Create a '`Make.<archname>`' in the '`hpl`' directory. Consider an `archname` like `Win2k_P4_CBLAS_p4` for a Windows 2000 system on Pentium 4 processors, using the C version of the BLAS constructed by ATLAS, and using the `ch_p4` device from the MPICH implementation of MPI. To create this file, look at the samples in the '`hpl/makes`' directory, for example,

```
% copy makes\Make.Linux_PII_CBLAS_gm Make.Win2k_P4_CBLAS_p4
```

Edit this file, changing `ARCH` to the name you selected (e.g., `Win2k_P4_CBLAS_p4`), and set `LAdir` to the location of the ATLAS libraries. Then do the following:

```
% make arch=<thename>
% cd bin\<thename>
% mpirun -np 4 xhpl.exe
```

Check the output to make sure that you have the right answer. The file '`HPL.dat`' controls the actual test parameters. The version of '`HPL.dat`' that comes with the `hpl` package is appropriate for testing `hpl`. To run `hpl` for performance requires

modifying '`HPL.dat`'. The file '`hpl/TUNING`' contains some hints on setting the values in this file for performance. Here are a few of the most important:

1. Change the problem size to a large value. Don't make it too large, however, since the total computational work grows as the cube of the problem size (doubling the problem size increases the amount of work by a factor of eight). Problem sizes of around 5,000–10,000 are reasonable.

2. Change the block size to a modest size. A block size of around 64 is a good place to start.

3. Change the processor decomposition and number of nodes to match your configuration. In most cases, you should try to keep the decomposition close to square (e.g., P and Q should be about the same value), with $P \geq Q$.

4. Experiment with different values for `RFACT` and `PFACT`. On some systems, these parameters can have a significant effect on performance. For one large cluster, setting both to `right` was preferable.

10.11 MPI-2 Status

MPI-2 is a significant extension of the MPI-1 standard. Unlike the MPI-1 standard, where complete implementations of the entire standard were available when the standard was released, complete implementations of all of MPI-2 have been slow in coming. As of June 2001, there are few complete implementations of MPI-2 and none for Beowulf clusters. Most MPI implementations include the MPI-IO routines, in large part because of the ROMIO implementation of these routines. Significant parts of MPI-2 are available, however, including the routines described in this book. Progress continues in both the completeness and performance of MPI-2 implementations, and we expect full MPI-2 implementations to appear in 2002.

10.12 MPI Routine Summary

This section provides a quick summary in C, Fortran, C++, and other MPI routines used in this chapter. Although these are only a small fraction of the routines available in MPI, they are sufficient for many applications.

C Routines.

int **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Request *request)

int **MPI_Wait**(MPI_Request *request, MPI_Status *status)

int **MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status)

int **MPI_Waitall**(int count, MPI_Request *array_of_requests,
MPI_Status *array_of_statuses)

int **MPI_Win_create**(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, MPI_Win *win)

int **MPI_Win_free**(MPI_Win *win)

int **MPI_Put**(void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)

int **MPI_Get**(void *origin_addr, int origin_count,MPI_Datatype origin_datatype,
int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)

int **MPI_Win_fence**(int assert, MPI_Win win)

int **MPI_File_open**(MPI_Comm comm, char *filename, int amode, MPI_Info info,
MPI_File *fh)

int **MPI_File_set_view**(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
MPI_Datatype filetype, char *datarep, MPI_Info info)

int **MPI_File_read**(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)

int **MPI_File_write**(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)

int **MPI_File_read_all**(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)

int **MPI_File_write_all**(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)

int **MPI_File_close**(MPI_File *fh)

int **MPI_Comm_spawn**(char *command, char *argv[], int maxprocs, MPI_Info info,
int root, MPI_Comm comm, MPI_Comm *intercomm,
int array_of_errcodes[])

int **MPI_Comm_get_parent**(MPI_Comm *parent)

Fortran routines.

MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, request, ierror

MPI_IRECV(buf, count, datatype, source, tag, comm, request,ierror)
<type> buf(*)
integer count, datatype, source, tag, comm, request, ierror

MPI_WAIT(request, status, ierror)
integer request,status(MPI_STATUS_SIZE), ierror

MPI_TEST(request, flag, status, ierror)
logical flag
integer request, status(MPI_STATUS_SIZE), ierror

MPI_WAITALL(count, array_of_requests, array_of_statuses,ierror)
integer count, array_of_requests(*),
array_of_statuses(MPI_STATUS_SIZE,*), ierror

MPI_WIN_CREATE(base, size, disp_unit, info, comm, win, ierror)
<type> base(*)
integer(kind=MPI_ADDRESS_KIND) size
integer disp_unit, info, comm, win, ierror

MPI_WIN_FREE(win, ierror)
integer win, ierror

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
target_count, target_datatype, win, ierror)
<type> origin_addr(*)
integer(kind=MPI_ADDRESS_KIND) target_disp

integer origin_count, origin_datatype, target_rank, target_count,
target_datatype, win, ierror

MPI_GET(origin_addr, origin_count, origin_datatype,target_rank, target_disp,
target_count, target_datatype, win, ierror)
<type> origin_addr(*)
integer(kind=MPI_ADDRESS_KIND) target_disp
integer origin_count, origin_datatype, target_rank, target_count,
target_datatype, win, ierror

MPI_WIN_FENCE(assert, win, ierror)
integer assert, win, ierror

MPI_FILE_OPEN(comm, filename, amode, info, fh, ierror)
character*(*) filename
integer comm, amode, info, fh, ierror

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info, ierror)
integer fh, etype, filetype, info, ierror
character*(*) datarep
integer(kind=MPI_OFFSET_KIND) disp

MPI_FILE_READ(fh, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

MPI_FILE_WRITE(fh, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

MPI_FILE_READ_ALL(fh, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

MPI_FILE_CLOSE(fh, ierror)
integer fh, ierror

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm,
array_of_errcodes, ierror)
character*(*) command, argv(*)
integer info, maxprocs, root, comm, intercomm, array_of_errcodes(*),
ierror

MPI_COMM_GET_PARENT(parent, ierror)
integer parent, ierror

C++ routines.

Request MPI::Comm::Isend(const void* buf, int count,
const Datatype& datatype, int dest, int tag) const

Request MPI::Comm::Irecv(void* buf, int count, const Datatype& datatype,
int source, int tag) const

void MPI::Request::Wait(Status& status)

void MPI::Request::Wait()

bool MPI::Request::Test(Status& status)

bool MPI::Request::Test()

void MPI::Request::Waitall(int count, Request array_of_requests[],
Status array_of_statuses[])

void MPI::Request::Waitall(int count, Request array_of_requests[])

MPI::Win MPI::Win::Create(const void* base, Aint size, int disp_unit,
const Info& info, const Intracomm& comm)

void MPI::Win::Free()

void MPI::Win::Put(const void* origin_addr, int
origin_count, const Datatype& origin_datatype, int target_rank, Aint
target_disp, int target_count, const Datatype& target_datatype) const

void MPI::Win::Get(void *origin_addr, int
origin_count, const MPI::Datatype& origin_datatype, int target_rank,
MPI::Aint target_disp, int target_count,
const MPI::Datatype& target_datatype) const

void MPI::Win::Fence(int assert) const

MPI::File MPI::File::Open(const MPI::Intracomm& comm, const char* filename,
int amode, const MPI::Info& info)

MPI::Offset MPI::File::Get_size const

void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
const MPI::Datatype& filetype, const char* datarep,
const MPI::Info& info)

void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype,
MPI::Status& status)

void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype)

void MPI::File::Write(void* buf, int count, const MPI::Datatype& datatype,
MPI::Status& status)

void MPI::File::Write(void* buf, int count, const MPI::Datatype& datatype)

void MPI::File::Read_all(void* buf, int count, const MPI::Datatype& datatype,
MPI::Status& status)

void MPI::File::Read_all(void* buf, int count, const MPI::Datatype& datatype)

void MPI::File::Write_all(const void* buf, int count,
const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Write_all(const void* buf, int count, const MPI::Datatype& datatype)

void MPI::File::Close

MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
const char* argv[], int maxprocs, const MPI::Info& info, int root,
int array_of_errcodes[]) const

MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
const char* argv[], int maxprocs, const MPI::Info& info, int root) const

MPI::Intercomm MPI::Comm::Get_parent()